ADDING PERSISTENCE TO MAIN MEMORY PROGRAMMING

A Thesis
Presented to
The Academic Faculty

by

Pradeep Fernando

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
December 2020

Copyright © 2020 by Pradeep Fernando

ADDING PERSISTENCE TO MAIN MEMORY PROGRAMMING

Approved by:

Professor Ada Gavrilovska, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Tushar Krishna
Electrical and Computer Engineering
Georgia Institute of Technology

Professor UmaKishore Ramachandran
School of Computer Science
Georgia Institute of Technology

Dr. Amitabha Roy
Google

Professor Joy Arulraj
School of Computer Science
Georgia Institute of Technology

Date Approved: 06/29/2020

To my Amma and Thaaththa

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude and appreciation to my advisor, Prof. Ada Gavrilovska, without whom this dissertation would not have been possible. Ada was instrumental in shaping up my research and career directions from the very beginning of my Ph.D. life. She gave me close guidance during the initial years of my Ph.D. life and helped me to develop a more refined taste in computer science research, gave me time, space and resources to grow myself as a capable researcher, and opened up additional opportunities for exciting research work/collaborations by introducing me to her fellow network of researchers. I am very privileged to be advised by her.

During my time at Georgia Tech, I had the opportunity to collaborate with some great researchers in computer systems. First, I thank them all in general for their positive impact on this dissertation work. I thank Dr. Greg Eisenhauer and the late Prof. Karsten Schwan for their help and guidance in tackling high-performance computing related research. Special thanks to my collaborator and friend Sudarsun Kannan for introducing me to the research topics around persistent memory programming. Little did I know that I would write a whole dissertation around the same. I thank Dr. Amitabha Roy and Dr. Subramanya Dulloor for their support and guidance on the fault-tolerant persistent memory work. I also thank Dr. Irina Calciu and Dr. Jayneel Gandhi for their collaboration and advice on the crash-sync persistent memory work. Additionally, I would like to thank the other members of my dissertation committee, Prof. Joy Arulraj, Prof. Kishore Ramachandran, and Prof. Tushar Krishna for serving on my committee and providing valuable suggestions/feedbacks for improving this dissertation work.

My time at Georgia Tech was made pleasant and enjoyable in large parts due to colleagues, friends, and mentors I met during my time in Atlanta. Special thanks to the fellow Georgia Tech student and my friend, Saminda Wijeratne, for helping me to transition into Atlanta life from my home country smoothly. I also thank many of my past and present

fellow Ph.D. colleagues in the CERCS lab for all the thought-provoking discussions we had and the time spent in general. A special mention to Dr. Ketan Bhardwaj, Anshuman Goswami, Alex Merritt, Thaleia-Dimitra Doudali, Chao Chen, Mohan Kumar, Sanidhya Kashyap, Ranjan Venkatesh, and Harshit Daga. Additionally, I would like to thank my graduate advisors Prof. H. Venkateswaran, Prof. S. Boldyreva, and our support staff, Susie McClain, Tiffany Ntuli for all the help during my time in Georgia Tech.

It would not have been possible to pursue higher studies without the support of my parents, wife, and sister. I thank my dear wife, Prabasha, for giving me love and encouragement throughout the time during this dissertation work. Finally, I thank my parents, whom I have the utmost gratitude towards (I miss you, dad). They sacrificed their whole life/time so that I could achieve my life goals.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Unlocking the true potential of the new persistent memories (PMEMs) requires eliminating traditional persistent I/O abstractions altogether, by introducing persistent semantics directly into main memory programming. Such a programming model elevates failure atomicity to a first-class application property in addition to in-memory data layout, concurrency-control, and fault tolerance, and therefore requires redesign of programming abstractions for both program correctness and maximum performance gains. To address these challenges, this thesis proposes a set of system software designs that integrate persistence with main memory programming, and makes the following contributions.

First, this thesis proposes a PMEM-aware I/O runtime, NVStream, that supports fast durable streaming I/O. NVStream uses a memory-based I/O interface that integrates with existing I/O data movement operations of an application to accelerate persistent data writes. NVStream carefully designs its persistent data storage layout and crash-consistent semantics to match both application and PMEM characteristics. Specifically, we leverage the streaming nature of I/O in HPC workflows, to benefit from using a log-structured PMEM storage engine design, that uses relaxed write orderings and append-only failure-atomic semantics to form strongly consistent application checkpoints. Furthermore, we identify that optimizing the I/O software stack exposes the PMEM bandwidth limitations as a bottleneck during parallel HPC I/O writes, and propose a novel data movement design – PHX. PHX uses alternative network data movement paths available in datacenters to ease up the bandwidth pressure on the PMEM memory interconnects, all while maintaining the correctness of the persistent data.

Next, the thesis explores the challenges and opportunities of using PMEM for true main memory persistent programming – a single data domain for both runtime and persistent application state. Such a programming model includes maintaining ACID properties during

each and every update to application's persistent structures. ACID-qualified persistent programming for multi-threaded applications is hard, as the programmer has to reason about both crash-consistency and synchronization – crash-sync – semantics for programming correctness. The thesis contributes new understanding of the correctness requirements for mixing different crash-consistent and synchronization protocols, characterizes the performance of different crash-sync realizations for different applications and hardware architectures, and draws actionable insights for future designs of PMEM systems.

Finally, the application state stored on node-local persistent memory is still vulnerable to catastrophic node failures. The thesis proposes a replicated persistent memory runtime, Blizzard, that supports truly fault tolerant, concurrent and persistent data-structure programming. Blizzard carefully integrates userspace networking with byte addressable PMEM for a fast, persistent memory replication runtime. The design also incorporates a replication-aware crash-sync protocol that supports consistent and concurrent updates on persistent data-structures. Blizzard offers applications the flexibility to use the data structures that best match their functional requirements, while offering better performance, and providing crucial reliability guarantees lacking from existing persistent memory runtimes.

CHAPTER I

INTRODUCTION

Emerging, byte-addressable persistent memory (PMEM) hardware technologies such as phase-change memory and STT-RAM have made significant progress in the last decade. Recently, the first large-capacity commercial products believed to be based on phase-change memory [1, 2] were released, targeting the server market. These new memories support byte-addressability via CPU load/store instructions, much like main memory (DRAM), but differ from the latter, as they hold the data state across node-restarts, similar to a persistent disk device. While the exact numbers depend on the PMEM hardware technology itself, the PMEMs in general, support high capacity memory modules because of their high density, have high access latencies, e.g., writes are up to $4\times$ slower than DRAM, and have limited device bandwidth, e.g., up to $8\times$ lower than the DRAM. However, poor capacity scaling of DRAM main memory, combined with modern applications' demand for fast, volatile and persistent storage, drive the integration of the PMEMs in compute platforms in datacenters [3, 4, 5] and in HPC exascale systems [6]. In these systems, PMEMs can be integrated in two main ways. In the first one, PMEM is placed "behind" DRAM, in a configuration where the system DRAM represents a hardware-managed cache for a much larger memory capacity provided by the PMEM device; this configuration does not exploit the persistent properties of PMEM. In the second, PMEM is placed "side-by-side" with DRAM, as illustrated in Figure 1. In this, more interesting, configuration, both the additional capacity and the persistence provided by the PMEM device are directly exposed to applications and software stacks. This thesis is primarily concerned with systems incorporating DRAM and PMEM in this "side-by-side" configuration, referred to as App Direct mode in the context of the Intel Optane DC Persistent Memory [1].

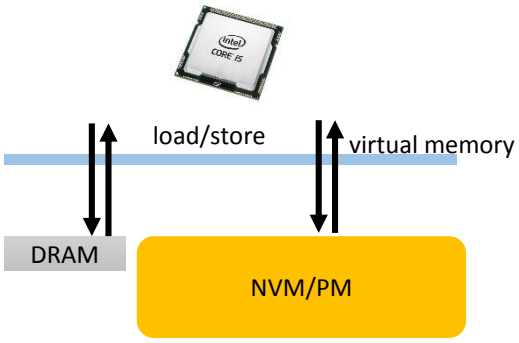

Figure 1: DRAM and PMEM memory configuration

Augmenting the memory hierarchy with PMEMs and leveraging the full benefits that PMEMs offer, presents both opportunities and challenges to system software designers. This is because the existing persistent system software stacks such as file systems and database management systems (DBMS) are built and optimized upon fundamentally different hardware abstraction – persistent block devices. On the other end of the spectrum, we have volatile memory programming abstractions – data-structures such as hashmaps, B-trees, etc., that permit direct manipulation of the memory words using load/store instructions. Main memory programming semantics on PMEM are shown to yield the maximum performance benefits [7] since it minimizes the imposed software instructions in the critical path of the PMEM accesses, however at the cost of program correctness. This is because, unlike DRAM memory, PMEM resident data survives across node restarts. An unplanned node restart is likely to terminate an ongoing sequence of PMEM updates midway, resulting in an inconsistent persistent program state. Furthermore, most main memory programming platforms integrate transient CPU caches for performance, that in turn delay and/or reorder memory updates back to the memory device, making it impossible to guarantee the correctness of the persistent data in the event of a crash. Solving this involves integrating crash-consistent semantics in to application programs as a first class citizen.

The system software research community has made significant progress in addressing both of these challenges, by redesigning PMEM optimized file-systems [8, 9], DBMSs [10]

and programming libraries [11] and data structures [12]. Still a number of important questions remain open. In the context of PMEM and modern applications, are the existing I/O runtimes and abstractions good enough for fast, durable data movements? How to support fast, main memory like persistent programming abstractions without compromising the program correctness? How such new abstractions interplay with concurrency control of the applications? Further, how to support truly fault tolerant persistent memory programming? We next discuss these questions in detail.

1.1 Statement of problems

- **Memory speed persistent I/O:** Persistent disk storage hardware, accessed using the I/O subsystem is incredibly slow compared to volatile main memory. Furthermore, applications often use software I/O intermediaries like file-systems and DBMSs to manage their persistent state. These system software stacks are very useful with disk storage as they accelerate data movements using buffering, manage concurrent accesses to data, provide consistent data semantics, etc. Most of the existing software falls in to this class of applications and thus it is important to accelerate their persistent data usage model with new PMEM hardware. The recent PMEM aware persistent I/O proposals [8, 9] address the above by redesigning system software internals while keeping the abstraction unchanged. However, the generic system design and the abstractions themselves are likely to limit further potential performance gains. One reason is that these traditional file system abstractions imply functionality (e.g., in terms of metadata management) or semantics (in terms of visibility of file system updates) that form significant portions of the I/O costs when used with fast PMEM devices; yet, they are not even necessary for important classes of I/O behaviors (such as for checkpoint generation and for streaming I/O in application workflows).

- **Reason about synchronization and crash-consistency in persistent programming:** Multi-threaded shared memory programming uses concurrency-control primitives

such as locking and transactional-memory (software/hardware) for program correctness. They provide atomicity, consistency and isolation (ACI) guarantees for multi-threaded programs. Crash-consistent persistent memory updates in the form of storage transactions, independently, also provide the same guarantees, plus durability (ACID). One way to simplify the development of PMEM applications is to use a single abstraction that provides guarantees on both crash consistency and correct synchronization. To this end, prior work has proposed numerous systems to simultaneously guarantee both these properties [11, 13, 14, 15, 16, 17, 18]. While these systems provide transactions with the desirable ACID properties that permit their use for both `crash-consistency` and `synchronization`, they do so in a myriad of ways, some implemented completely in software while some rely on hardware support, some use undo logging, while others use redo logging. So, developers are faced with a bewildering array of choices, with varied performance characteristics that change with applications and the system used. For a platform with given hardware features and consistency and synchronization requirements, is it possible to streamline the design space and quickly arrive at a correct and performant implementation of a transactional system?

- **Truly fault tolerant persistent memory:** Commodity servers which integrate PMEM provide data reliability by means of hardware (erasure-codes) and software (RAID) mechanisms, but they do not tolerate full node failures. A truly fault tolerant persistent memory programming requires some form of data redundancy across nodes. However, it is not straight forward to add fault-tolerant semantics to persistent data structures while supporting concurrent, crash-consistent updates. This is because, each of these semantics have their unique and often conflicting system software trade-offs between each other. For an example, replication stacks are often designed with serial log commits for correctness while persistent data-structures use concurrent updates for maximum throughput.

Summarizing, simply using memory-based interfaces and programming models to maximize the performance of PMEM-based applications and software stacks in not trivial, and presents a number of challenges that must be addressed, and tradeoffs that must be fully evaluated in order to be leveraged.

1.2 Thesis statement

It is possible to make existing application-level memory data-structures and programming interfaces persistent and to thereby deliver better functionality and performance, compared to efforts optimizing I/O storage abstractions for PMEM.

To support the hypothesis, this dissertation makes the contributions outlined below.

1.3 Contributions

- **Memory speed persistent I/O:** This thesis proposes a memory centric object-based abstraction for persistent I/O. The supporting system – NVStream – comprises an I/O library and runtime system, and its design is specialized for data movement in analytics workflows and checkpoint I/O, such as what is used in HPC. The design of NVStream combines a streaming, versioned object-store, with efficient log-based memory management, and hardware-accelerated persistence for consumer-producer patterns. The memory friendly persistent I/O API supports data movement and persist optimizations that uses both hardware and domain specific application characteristics.

- **PMEM bandwidth aware persistent I/O:** Removing the software overheads from the PMEM access paths, exposes new bottlenecks. For memory-based checkpoint I/O in HPC applications, where many threads concurrently issue coordinated checkpoint operations, the limited PMEM bandwidth poses a significant challenge in realizing the expected benefits from using PMEMs.

We propose a PMEM bandwidth aware persistent I/O library (PHX) for HPC checkpoint I/O. PHX deals with the limited PMEM bandwidth through simultaneous use of PMEM and local/ peer nodes' DRAM devices, thus increasing the effective data movement bandwidth. PHX's memory-centric object interface and PMEM-bandwidth-aware design lead to reduction in the time length of I/O operations in the critical path, associated with the slow PMEM device. To continue guaranteeing adequate reliability and persistence, DRAM-resident object state is replicated across peer nodes' memory, which is accessible through high-bandwidth interconnects. The use of a memory-based I/O both exposes the problem (since it would not be an issue if applications used "longer" block-device based I/O paths) and enables the solution (by making it possible to realize a low-cost replication so that the additional data movement introduced by PHX is outweight by substantial gains in application performance and system efficiency).

- **Building persistent memory systems with crash-sync-safety:** We characterize different transaction systems to identify the design space of ACID transactions. The candidates are chosen based on *crash-sync-safety* property, that guarantee 1) proper synchronization, 2) crash-consistency semantics, and 3) correct composibility of (1) and (2). This new characterization of transaction systems provides a basis to compare different implementations and to identify the right set of crash-consistency and synchronization mechanisms for particular applications and hardware platforms.

- **Fault-tolerant persistent memory:** This thesis proposes a system software solution – Blizzard – that supports highly performant and fault-tolerant (replicated) persistent memory programming. The key idea in Blizzard is to exploit direct accesses to persistent memory from both CPU and the commodity NIC to provide efficient zero copy replication of RPC calls. We build on this by providing a library of persistent data-structures with a recipe for concurrency that works well with the replication.

1.4 Organization

The remainder of this dissertation is organized as follows. In Chapter 2 (§2) we give a brief introduction to byte-addressable persistent memory (PMEM). We discuss the properties of PMEM hardware, usage models and introduce the importance of ACID qualified PMEM programming.

Chapter 3 (§3) details `NVStream`, a PMEM aware system software stack for streaming I/O use cases in high performance computing application workflows. Also, the chapter includes an evaluation of `NVStream` against three real-world HPC applications, GTC, CM1, and S3D.

Chapter 4 (§4) presents `PHX`, that leverages alternate interconnects and memory resources to alleviate PMEM bandwidth bottleneck during HPC checkpoint I/O. The chapter also includes `PHX` evaluations against three HPC applications, GTC, CM1, and miniAMR.

Chapter 5 (§5) characterizes properly synchronized and crash-consistent, transactional programming on PMEM on platforms with different hardware properties and with different application use-cases. We detail the lessons learned and draw insights for future PMEM transactional programming system selection and designs.

In Chapter 6 (§6), we detail `Blizzard`, a runtime for highly available PMEM programming. We model and evaluate durable enterprise application backends using `Blizzard`'s memory native persistent data-structures.

In Chapter 7 (§7), we discuss the other significant research work related to this thesis and put our work in context. And finally, in Chapter 8 (§8) we discuss and conclude this dissertation with ideas for possible future research opportunities.

CHAPTER II

BACKGROUND

In this chapter, we introduce byte-addressable persistent memory (PMEM) hardware and put it in the context by comparing it against other related memory and persistent media device characteristics. PMEM hardware supports multiple usage models when integrating with applications.

2.1 Background on persistent memory

The well-known DRAM main memory realizes binary states using capacitor stored temporary electrical energy. DRAM main memory therefore is volatile and notorious for weak capacity scaling, as the device energy consumption is proportional to memory capacity. Byte-addressable persistent memory or simply persistent memory (PMEM), on the other hand realizes binary state using changing the material resistance using an electrical charge. The persistent memory updates are slow and consume more energy due to this technical difference. However, they have excellent capacity scaling, as the device does not need energy refresh cycles. More importantly, for the same reason the written data is persistent and survives across machine restarts. Table 1 compares [19] device properties of PM against DRAM main memory and SSD/HDD storage.

Table 1: Comparison of PCM based PMEM with DRAM and other storage devices. Data derived from [19] and experiments on Intel Optance DC PMEM

	DRAM	**PCM**	**SSD**	**HDD**
Capacity per CPU	100s of GBs	Terabytes	Terabytes	Terabytes
latency	60 ns	300 ns	300 µs	10 ms
bandwidth	1×	$\frac{1}{4}\times to \frac{1}{8}\times$	-	-
Addressability	Byte	Byte	Block	Block

2.1.1 Persistent memory usage models

While it is possible to fully replace DRAM main memory with the new persistent memory modules, thus dual-purposing PMEM for both volatile and persistent memory use-case, we are likely to have both of these memory options available in the future hardware configurations. PMEM accesses are still costly relative to $\sim 60ns$ DRAM access latencies and therefore having DRAM capacity benefits applications with high frequency, but volatile runtime state maintenance. Confirming our observation, the current generation commercial PMEM offerings, specifically based on Intel Optane DC integrates both DRAM and PMEM support in their hardware design. They integrate PMEM by attaching it to the memory bus using a separate memory controller, in a way that makes it appear as a non-uniform memory access (NUMA) memory node. Furthermore, the hardware comes with two major DRAM/PMEM configuration options – Memory Mode and App-Direct Mode. The former, treats PMEM as a volatile memory device front-ended by a hardware managed DRAM cache. The latter, configures DRAM and PMEM in a side-by-side setting, where applications have total control over the PMEM address space. This thesis assumes PMEM equipped machines configured in the App-Direct Mode, as the configuration supports both persistent PMEM usage and PMEM system software design. In App-Direct Mode, a file system manages the PMEM address space and the approach is consistent with the Linux's use of file names as resource handles. Applications can perform file I/O or direct memory load/stores on app-direct configured PMEM device.

Legacy applications using block I/O (e.g., file I/O, DBMSs, key-value stores) for their persistent data storage can use the same APIs to store data on a PMEM device without any additional program changes. Instead, PMEM-aware block I/O system software stacks [8, 9, 20, 10] optimize their system software internals to deliver PMEM performance advantage to end-user applications. These systems take advantage of byte-addressable persistence by removing outdated system components [8], re-designing internal data-structures and protocols to match the capabilities of new hardware, etc.

In direct memory PMEM programming, applications bypass the file-system software altogether and make direct changes to the data on PMEM using load/store CPU instructions. Applications map PMEM memory region as a file mapping using regular `mmap` system call and direct-memory-access (DAX) supporting PMEM aware file systems route load/store to the PMEM device without software intervention. PMDK [21] like programming tool-kits Programming toolkits such as Intel's Persistent Memory Development Kit (PMDK) provide convenience library support to handle common PMEM programming tasks, including PMEM memory region management, offset based persistent pointers and failure-atomic PMEM programming semantics (PMEM transactions).

2.1.2 Crash-consistent persistent memory programming

PMEM is directly accessible by CPU instructions (e.g. load/store), and direct programming of persistent state without software intermediaries such as file-systems is one of the most desirable properties of this new type of devices. However, direct programming persistent memory involves maintaining consistent persistent data state at all times using careful data-update sequences – crash-consistent update protocols. Crash-consistent protocols enforce atomic, consistent, isolated and durable (ACID) data updates on PMEM and protect data corruption during unplanned node shutdowns.

Intel x86 architecture CPUs only support 8 byte (memory aligned and contiguous) atomic memory updates. Therefore, PMEM atomic updates involving multiple 8 byte blocks are not natively supported for atomicity in hardware. PMEM optimized software crash-consistency protocols extend the basic hardware primitives to support more sophisticated PMEM update sequences. Crash-consistency protocols optimized for PMEM that use write-ahead-logging protocols (e.g., undo/redo logging) [11] and copy-on-write (COW) [10] are widely used for this purpose.

2.1.3 Commercial availability of PMEM

The earliest realization of PMEM hardware was to combine volatile DRAM memory module with an on-device emergency power source – battery-backed DRAM. In an unplanned node shutdown (e.g., power failure), these PMEM devices use the limited emergency power to write out DRAM memory resident data on to a block device(e.g., SSD). Battery-backed DRAMs offer byte addressable persistence at DRAM memory speeds but has very poor capacity scaling. In addition to memory refresh cycles, battery-backed DRAM based PMEMs have to support sufficient amount of emergency power to safely backup all the volatile memory content. Additionally, battery-backed DRAM configured PMEMs often backs-up all the volatile memory state including CPU caches (persistent CPU caches).

More recently, Intel debuted their 3DXPoint [22] based PMEM hardware and the device characteristics matches the PCM based PMEM numbers we quote in Table 1. These are high-capacity memory devices and each PMEM memory module can support up to 256GB. However, L1-L3 CPU caches remain transient/volatile as the persistence domain starts at the memory controller. Transient CPU caches with write-back caching policies complicate PMEM programming, as these caches both buffer and re-order the PMEM stores before they reach the memory controller/persistent domain. PMEM aware crash-consistency protocols solve this problem by carefully ordering the PMEM writes and explicitly flushing CPU cachelines. New generation x86 CPU hardware introduces optimized cache-line flushing instructions `clflushopt, clwb` to offset some of the overheads associated with crash-consistent PMEM updates.

2.2 Chapter summary

Byte-addressable persistence makes new PMEM hardware attractive for integration with memory-based interfaces for application programming and even I/O, but the significant differences among PMEM and DRAM and across PMEM technologies in terms of their performance, capacity, access and failure models, make this a non-trivial process. The next

four chapters further illustrate the specific nuances of how PMEM is used and operates, and presents the system solutions needed to best leverage the opportunities provided by PMEM while avoiding or reducing the impact of the inherent limitations of these technologies.

CHAPTER III

MEMORY ACCELERATED PERSISTENT I/O

Recent systems research has explored ways of improving the traditional application I/O stacks with new PMEM device capabilities. These efforts include PMEM optimized file-system implementations [8, 9, 23] with shortened I/O path distances (e.g., page cache removal), memory-optimized indexing and memory-optimized crash-consistency protocols. Still, these PMEM optimized storage stacks perform poorly for high-performance computing (HPC) I/O use-cases. In this chapter, we discuss software limitations during HPC-I/O in detail and present a system software solution to address them.

3.1 Introduction

Long-running scientific computations, such as material combustion, fusion and climate modeling, periodically produce the program outputs of the simulation state. These periodic program outputs serve multiple purposes. First, they are used as an application checkpoints, which are used for recovery in the event of simulation or system failure. Second, they are increasingly being used to provide online insights into the simulation state, and are directly consumed by co-running coupled analytics programs, performing output visualization, verification, uncertainty analysis, or other data analysis tasks [24, 25]. This producer-consumer relationship among application components establishes a streaming workflow and HPC technologies supporting streaming workflows are becoming an integral part of HPC systems [26, 27, 28].

Although at a high-level, streaming workflows provide an opportunity for analytics application to quickly consume simulation data, the performance of the streaming infrastructure is highly dependent on the performance and capacity of the available memory and storage resources. This is because, in streaming workflows, in-memory buffers are used

to temporarily stage simulation outputs that must be written to storage (e.g., for checkpointing), or for a co-running workflow component to consume the data. However, due to significantly large data volumes and the limited amount of DRAM capacity, the volume, and frequency of data that can be generated and passed through the workflow components is restricted. To overcome these challenges, production-level systems, such as ADIOS [26] use persistent storage (flash, hard-disk) devices and OS-level file systems for temporarily staging data, thereby overcoming the DRAM capacity limitations. Because the data movement across workflows is used for both reliability (i.e., checkpointing) and for coupled analytics, it must be carried out without sacrificing reliability or consistency properties. As a result, these systems typically rely on the underlying file system crash-consistency and durability mechanisms such as journaling and logging.

However, using slower storage devices such as flash or hard disk to stage data as a replacement for DRAM can significantly increase data exchange cost in streaming workflow due to significantly higher latency and lower bandwidth compared to using DRAMs. Emerging non-volatile memory technologies such as 3D-Xpoint DIMMs [22] provide 100$\times$ faster read/writes and up to 10$\times$ higher bandwidth compared to flash memory, and can accelerate the data exchange performance. Unfortunately, naive use of PMEM underneath file system stacks can substantially limit PMEMs hardware benefits provided to streaming workflows. This is because the traditional file system based solutions incur overheads in several ways, including high serialization (memory to block conversion) overheads, deserialization overheads, and POSIX-based system call and journaling/logging costs in the data movement path, which is not necessary given the byte-addressable interface exposed by PMEM. Even file systems designed explicitly for PMEM [8] continue to impose these undue software overheads. Furthermore, adherence to full file system semantics prevents software from taking advantage of new PMEM-centric architectural features, such as support for streaming, non-temporal writes [29], which are particularly useful for streaming

workflows. Note that using memory copy-based operations with PMEM [30] is also inadequate: First, staged data poses requirements for temporal durability (i.e., to provide guaranteed delivery). Second, for correctness, the persistent nature of PMEM requires transactional mechanisms so that partial updates are not persisted and delivered to downstream consumers.

Based on these observations, we design and implement `NVStream`, a user-level transport for workflow coupling and high-performance data streaming via PMEM. `NVStream` accelerates HPC streaming I/O by leveraging the memory-based nature of PMEM, the streaming semantics and temporal durability requirements of scientific workflow systems, and the new architectural capabilities in modern processor architectures. Its design combines streaming, versioned object-store, with efficient log-based memory management, and hardware-accelerated persistence for consumer-producer patterns. Additional optimizations in the data movement path for applications exhibiting temporal locality in their data access behavior is realized through the use of delta encoding [31].

In this chapter;

1. We provide an in-depth analysis of the characteristics of using PMEM-based transport channels in streaming workflows.

2. We design a novel solution for streaming persistence which accelerates PMEM-based producer-consumer data exchanges.

3. We design and develop `NVStream`, a new PMEM-specific system for coupling in-situ analytics in HPC systems based on streaming persistence.

The current `NVStream` implementation focuses on a single-node, multi-core, shared memory platform and we evaluate `NVStream` with several data-intensive benchmarks and HPC applications that are running on emulated PMEM.

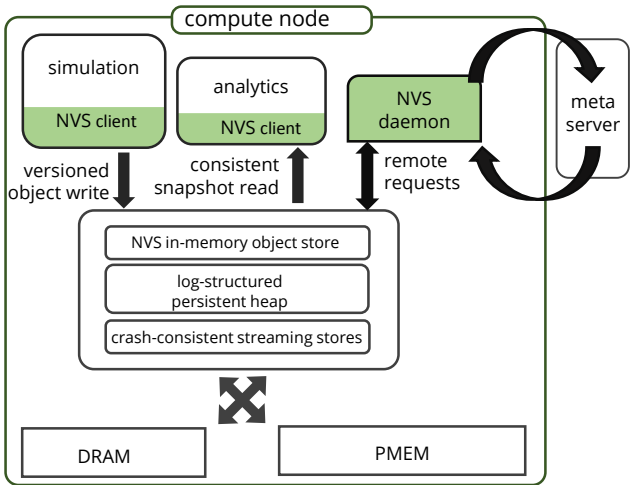

Figure 2: The high level component design of `NVStream`.

3.2 *`NVStream` Overview*

We design `NVStream`, a PMEM-aware data transport for HPC workflow I/O. The main goals of the `NVStream` are:

- HPC data transport for node-local and cross-node workflow interactions;
- persistent and consistent simulation output state maintenance;
- PMEM device-aware data transport design; and
- exploiting HPC application I/O awareness for high performant transport design.

We first briefly describe a high-level usage of the proposed system in the context of HPC workflow I/O, followed by a deep dive into each of the components of the system.

HPC workflow components, i.e., data producers and consumers, interact with the `NVStream` runtime using a set of I/O APIs. These APIs are modeled after familiar memory like API calls. We determined that the effort of porting a traditional file I/O-based application to `NVStream` is minimal. For an example, we ported the GTC and CM1 applications to `NVStream` with $\sim$ 15 and $\sim$ 85 number of source code line updates.

Figure 2 shows the control and data path interactions between workflow producers and consumers with the `NVStream` runtime. The application that generates data, such as the

```
nvs_init(store_id); /* runtime init */
/* object allocation with the key = "zeta" */
void *obj = nvs_create_obj("zeta", sizeof(int)*1000);
int *array = (int *)obj;/* casting to our own type */
/*
 * perform computation steps on this object
 */
nvs_put("zeta"); /* output write */
nvs_free_obj{"zeta"};
nvs_finalize(); /* runtime shutdown */
```

Figure 3: Science simulation producers producing program snapshots using `NVStream` API

simulation code, first declares the output variables by allocating them using `NVStream` API call `nvs_create_obj`, as shown in Figure 3. The call registers the allocation metadata with the `NVStream` runtime and returns a contiguous volatile memory block similar to `malloc`. Later, the producer outputs a versioned snapshot of the object using `nvs_put`, which stores the snapshot durably on the underlying PMEM-based shared memory layer. The consumer application in the workflow uses similar calls to retrieve consistent snapshot versions from the `NVStream` transport channel.

The major components in the `NVStream` runtime which support these interactions are:

Log-structured heap is a PMEM-aware shared heap implementation, responsible for allocating space on the shared PMEM memory and providing for crash-consistent data writes and reads. §3.3

Object-store is the application-facing component of `NVStream`. It exposes a set of APIs to both producers and consumers for workflow interaction. §3.4

NVStream daemon is responsible for background services such as garbage collection of old `NVStream` objects and support for remote/cross-node data transfers. §3.5

3.3 Log-structured Memory Heap

3.3.1 Persistent Smart Pointers

NVStream I/O transport uses persistent shared memory to move data between producers and consumers in a coupled workflow. We manage persistent shared memory regions as named mappings (mmap call) supported by the OS. Each mapping is uniquely identified as a file name – we identify it as map_id. A process independent unique shared memory address is constructed using <map_id :offset> where offset denotes the number of byte offset value from the beginning of mapped segment's start address. Given a shared memory address, a process accessing persistent shared memory data, first maps the memory segment identified by map_id into its own address space and then records the starting virtual memory address of the mapped segment – map_start. Next, the process local virtual address is obtained by {map_start + offset}.

We need a similar addressing mechanism for structures placed on the volatile shared memory as well. We use boost [32] interprocess primitives for the same.

3.3.2 Shared Persistent Heap

NVStream stores the producer generated data and metadata on PMEM before they are streamed to the workflow consumer/s. For efficient data updates/lookup, the data writes on persistent devices follows a pre-determined data storage layout/format. The system software stack operating on the persistence device often determines the specifics of the data layout. For example, pmfs formats the PMEM device space with a B-tree index structure for efficient file-data traversal. While complex data layouts such as inodes and B-tree indexes improve on-device data organization and data traversal speeds, they often come at a cost in the form of excessive metadata overheads, costly crash-consistency routines, etc.

We learn that the simulation program snapshots, placed on the NVStream persistent memory regions are versioned/immutable objects. These immutable objects are consumed by one or more downstream analytics applications – often in a monotonically increasing

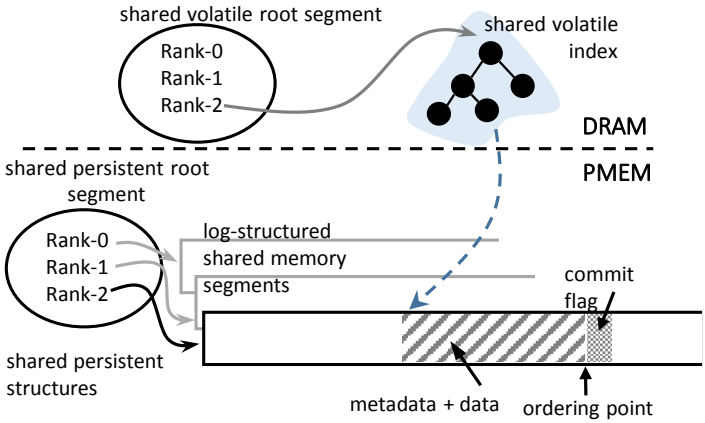

Figure 4: `NVStream` stores data in a shared memory log-structured heap formatted on PMEM device and maintains volatile state of the runtime, in DRAM. The indexing structures are placed in shared volatile memory for multi-process accessibility

object version order. The immutable objects exchanged between producers and consumers form a streaming data pattern and is well-supported by the FIFO data structure.

The above observation motivates us to format (see §4) the `NVStream` persistent shared memory as a log-structured append-only storage heap (heap-log). The new data gets appended to the tail of the log (marked by `log_tail`) while old data gets taken out from the log-head (marked by `log_head`. The heap-log is generic enough that it does not enforce any layout on its append entries. The high-level software abstractions, such as the `NVStream` object-store, defines the specific entry format that describes the stored data.

In addition, we acknowledge the fact that almost all HPC applications follow single-program multiple-data compute pattern influenced by MPI parallelization framework and thus perform parallel I/O. We optimize the `NVStream` persistent heap for the same by maintaining per-process heap-log – supporting efficient parallel data writes. A root persistent object act as the entry point to the `NVStream`'s storage hierarchy and is uniquely identified by a `store_name`. It keeps track of the heap-logs – identified by their `rank` id.

Our design choice of formatting PMEM device as a log-structured heap greatly simplifies the crash-consistent data updates on PMEM and more importantly, enables high throughput persistent data writes on to the PMEM device.

3.3.3 Crash-Consistent Data Streaming

Although, the data stored on PMEM device survives application and node restarts, maintaining metadata and data consistency of the stored data in the wake of unplanned crashes is a challenging problem. Current crash-consistent update techniques solve the problem by either atomically updating data or encoding enough information so that during recovery, one can reason about the consistent state. For designing crash consistent updates for `NVStream`'s persistent heap-log, we make the following observations:

Observation-1 - the versioned objects and the append-only log-structured heap do not mutate existing data/metadata during output writes.

Observation-2 - data producing HPC applications do not read-back the written out data in the fast path – happens only during a restart.

Observation-1 leads us to incorporate a logging-based crash-consistent updates – a natural design choice for our log-structured heap. To that end, crash-consistent and durable log appends to the heap-log includes following steps: ① acquire the write lock of the heap-log, ② issue writes that comprise of the appending data, ③ ensure all the writes have been committed to PMEM, ④ issue `commit` flag append to the heap-log, ⑤ ensure that `commit` flag write has also been committed to the PMEM device.

The regular `temporal-store` instructions provided by x86 processors operate on write-back(WB) mapped memory segments. They are optimized for temporal data access patterns and are buffered in the processor cache hierarchy before being evicted to the backing memory (write-back operation). Therefore, temporal-stores, when used for heap-log appends, requires explicit cacheline flushing (using `clflush, clflushopt`) to move cached temporal stores into the backing memory device. Cacheline flushing is expensive and significantly reduces the memory parallelism and bandwidth usage due to the ordered execution of flush instructions. In contrast, x86 streaming stores operate on write-through(WT) mapped memory segments, and these writes bypass the processor cache altogether. In addition, the streaming stores enjoy a relaxed memory consistency semantics in contrast to

total-store-ordering(TSO) memory consistency enforced on temporal-stores by the processor. Further, the absence of TSO and write-combining buffer support of CPU (batching) allows streaming stores to move data fast into the backing memory device. Interestingly enough, Observation-2 suggests that temporal stores are of little use to HPC applications, and importantly, temporal store-based I/O may negatively contribute to the producers compute performance as they are likely to evict the actual application's (simulation's) working-set from the cache.

Based on these observations, we use streaming stores for `NVStream`'s heap-log appends. We use store (`sfence`) instruction to enforce explicit ordering between streaming stores (e.g., steps ③ and ⑤ in the heap log append).

3.4 NVStream Object Store

3.4.1 Object Allocation

Application code allocates objects using `nvs_create_obj` API call. The API call is similar to `malloc` memory allocation call, but accepts additional `object_key` parameter. The `object_key` serves two purposes, 1) allow subsequent `NVStream` APIs to use it as a volatile handle for the allocated object (e.g., `nvs_put` and `nvs_free`), 2) provide analytics applications a persistent object handle to retrieve PMEM stored objects (e.g., `nvs_get`). The `NVStream` runtime in its internal representation maintains a fully qualified object handle in the form of [`store_id : object_key`] using the `store_id` given in `nvs_init`. Our prototype currently supports hierarchical `store_id`'s up to two levels– usually in the form of [`store_name/rank`]. For an example, an object with key, '`zeta`' that belongs to the MPI rank 1 of a coupled simulation named `gtc-analyze` is represented as [`gtc-analyze/1`:zeta].

`NVStream` object allocation involves allocating a memory chunk from the DRAM main memory (Linux glibc allocator) and updating the volatile index structure that maps the `object_key` to a memory address along with other metadata that includes object's size

and the current version. The compute kernel uses the returned object for regular/volatile reads and writes. During program snapshots, the `NVStream` runtime generates a versioned copy of this object on the persistent memory. We describe this operation next.

3.4.2 Object Write

`nvs_put` call creates a versioned persistent snapshot of the volatile object identified by the `object_key`. The call implements the following control sequence. First, `NVStream` runtime scans its internal structures to find the object address for a given key which was mapped and recorded during the object allocation request. Next, it makes a snapshot of the DRAM resident object on persistent memory by appending/copying object data into the process local heap-log (see Figure 4) along with name, version, and other metadata information of the object. Finally, we update the DRAM resident indexing structures with the newly written object information.

Batched appends. Most HPC applications perform I/O in batches; after a certain number of iterations, an application writes out all the intermediate objects/variables in a batch. We exploit this characteristic of HPC applications and introduce a batched object write API – `nvs_snapshot`. The `nvs_snapshot` call converts multi-object writes into one single append operation on the PMEM based heap-log. Batching the writes eliminates multiple crash-consistent write sequences that involve costly store ordering, and increases the memory bus bandwidth usage during data movement.

Delta-compression. Matrix-related arithmetic is a norm in HPC applications, and checkpoint/analytics output state mostly comprise of these matrix variable states. However, we learn that some HPC applications do not modify all the data points in their application matrices/grids across iterations. This observation opens up an opportunity to selectively store the data that has been changed since the previous I/O step – storing only the delta from the previous state. This optimization is called **delta-compression** and the `NVStream` flavor that integrates the optimization is NVSD.

The proposed NVSD uses memory page-based write protection support available in the memory management unit (MMU) and the OS exposed system call interfaces to create lightweight deltas in the userspace. The NVSD approach calculates deltas at object granularity and we page-align the object allocations. During each compute iteration after `nvs_put` operation, the NVSD runtime write-protects all the pages of one or more DRAM resident objects. As a consequence, any subsequent write to the memory protected object raises a page protection signal (`SIGSEG_FAULT`) leading to the following control sequences: 1) NVSD registered signal handler gets invoked, 2) the runtime determines the memory page and the object that belongs to the faulting memory address, and finally, 3) the modified/written pages are recorded in a bit vector and the write protection on the faulted memory page is removed. We use the modified page bit vector to calculate the delta of that particular object and only store the modified memory pages in heap-log. We store additional metadata information such as the relative position of the modified pages within an object; this information is used for reconstructing the exact object sate at the analytics application (the consumer) as if there were no delta-compression on the stored data.

3.4.3 Object Read

The `nvs_get` call retrieves a PMEM-resident persistent object and is internally implemented as follows. We first find the heap-log corresponding to the given [`store_id : object_key`] using the `rank` value in the `store_id`; if the `rank` belongs to local store, the requested object version is available on one of the heap-logs available on the local node/persistent device, else it has to be retrieved from a remote node. We look at each of those scenarios next.

Scenario1: ***object is local.*** We walk the rest of the volatile index structure in-search for [`object_key : version`] mapping. A successful search returns a persistent shared memory address [`map_id : offset`] of the heap-log stored object. The address uniquely identifies a persistent memory location in which the object resides using which we map the

corresponding log-heap into our own address space and read the object data/metadata at `offset`.

Scenario2: ***object is remote.*** The `NVStream` daemon process is responsible for retrieving remote objects over the network. We place a remote object retrieval request on the `NVStream` daemon's §3.5 shared memory request queue and synchronously wait for the response.

3.5 NVStream Daemon

3.5.1 Remote Data

The `NVStream` daemon component is responsible for 1) serving requests for local objects from remote nodes across the network, 2) fetching the objects located in remote nodes for local analytics processes, and finally, 3) for services such as garbage collection of stale objects.

Remote requests. The `NVStream` daemon receives object requests from remote peer nodes through a TCP server socket. Similarly, it looks-up the requested object by scanning the shared memory volatile index structure similar to that of the `nvs_get`. After an object is located, the daemon process serves it to the remote peer.

Remote objects. The `NVStream` daemon receives the remote object retrieval requests from local analytics processes through a shared memory queue structure. First, it finds the remote peer node that hosts the requested object; note that the object-to-remote host mapping is done via a metadata server hosted at a known address and the metadata server lookup maps the `store_id:rank` portion of a `object_key` to a hosting server address. Next, we synchronously request the object/s from the remote host. Finally, we serve the retrieved object to the local analytics process. It is important to note that the 'remoteness' of the requested object is transparent to the analytics application and is completely handled by the `NVStream` library/runtime. We cache the `store_name:rank` to `host-address` mappings for future interactions.

3.5.2 Garbage Collection

NVStream garbage-collection (GC) service cleans up the old versions of the objects from the heap-log. NVStream implements a simple GC algorithm called max_version_gc; the max_version_gc garbage collector purges all the stale object versions except most recent version (object with MAX version number). The GC service monitors the head and tail values of the heap-log and triggers garbage collection routine if the log space is filled up to a threshold value, which is a configurable parameter.

The NVStream's garbage collection routine steps include 1) calculating the new head offset of persistent log heap that constitutes the most recent MAX versions of objects, and 2) truncating the persistent log up to the new log offset.

The integration of delta-compressed objects into NVStream increases the complexity of log truncation operation; this is because the recent delta-based object versions may depend on the older versions of the same object for valid reconstruction of their complete object state. Thus simply truncating heap-log may result in losing some of the data parts which are critical for reconstructing more recent object versions out from their deltas. During NVSD heap-log truncation, we create a full program checkpoint that includes the objects of version MAX, thereby effectively truncating the dependency list of object deltas at the MAX object version. We maintain the MAX versioned program checkpoint separate from the heap-log. We only have to maintain one program snapshot version (the most recent one) per heap-log for guaranteeing correct reconstruction of full object state.

3.5.3 Failure Recovery

Recovering from failure includes restoring consistent runtime/volatile and persistent state of the system. During recovery, the NVStream daemon §3.5 re-initializes the shared volatile memory structures that include synchronization structures and metadata indexes. Populating volatile indexes for persistent objects involves walking the heap-log from log_head to

log_tail and updating volatile indexes corresponding to objects details found on the heap-log. Because of log-structured persistent heap design, the recovery steps of NVStream is almost trivial in contrast to WAL based crash-consistency designs, that involves replaying the undo/redo log for restoring the consistent persistent state.

3.6 Evaluation

We evaluate NVStream against several real-world and proxy HPC applications, 1) GTC, 2) CM1, and 3) miniAMR. Our experiments are designed to answer the following questions:

- How NVStream compares to state-of-the-art PMEM storage stacks for streaming data writes and reads?
- What is the effectiveness of NVStream's delta compression technique and its contribution to fast data movement?
- Does NVStream reduces overall application execution time in the context of workflows with analytics or checkpoint I/O?

3.6.1 Methodology and Benchmarks

Table 2: Node configuration of the in-house Aries cluster

Interconnect	Mellanox Infiniband
CPU core	Intel XeonE5 1.8GHz
CPU cores per node	80 cores over 4 dies
Main memory per node	500GB over 4 NUMA nodes

We emulate persistent PMEM using memory regions mapped over a DAX-enabled file system (pmfs) similar to [33]. DAX-enabled file systems allow direct load/store access to the underlying mapped memory (DRAM in this case), thus emulating the load/store interface of PMEM. Next, write-through memory mappings are not available in userspace as the Linux OS always maps the memory segments as write-back memory; therefore we use streaming stores over WB memory similar to [34]. Further, PMEM devices are expected to have low device bandwidth compared to DRAM main memory; thus the NVStream data

movement time depends on the PMEM device properties. Our current PMEM emulation platform assumes the device to have latency and bandwidth parameters similar to that of DRAM, and differentiate them only based on their data persistence support. We leave detailed sensitivity analysis of `NVStream` against different PMEM latency and bandwidth parameters for future work. Finally, all our experiments are performed on a local machine, Aries (see Table 2); a local testbed provides us with greater flexibility of software stack configurations (e.g., installing pmfs). Next, we briefly describe each of the main simulation applications – GTC, CM1, and miniAMR.

Gyrokinetic Toroidal Code (GTC) is a three-dimensional particle-in-cell application [35] used in micro-turbulence fusion device studies. The checkpoint data constitutes of 2D/3D arrays. We change the original input parameters 'npartdom', 'micell' and 'mecell' in constant factors to weak scale the workload size of the benchmark.

CM1 is a three-dimensional, non-hydrostatic, non-linear, time-dependent numerical model designed for idealized studies of atmospheric phenomena [36]. It is designed for studies of relatively small-scale processes in the Earth's atmosphere, such as thunderstorms. We change the input parameters `nx`, `ny` and `nz` to vary the workload size of the benchmark.

miniAMR applies a seven-point stencil calculation on a unit cube computational domain, which is divided into blocks. The blocks all have the same number of cells in each direction and communicate ghost values with neighboring blocks. With adaptive mesh refinement, the blocks can represent different levels of refinement in the larger mesh.

We use several alternative I/O mechanisms as comparison points for `NVStream`. We describe each of those I/O techniques next:

nvs: is the proposed `NVStream` implementation that uses persistent streaming writes for data movement.

nvs+delta: is the NVSD implementation that incorporates the delta compression technique into `NVStream`.

memcpy: is a best case user-space data movement mechanism for PMEM which does not

support crash-consistent data movements. We use the default `memcpy` routines supported by glibc.

tmpfs: is a volatile main memory-based file system which lacks crash-consistent data storage. The file system uses a ram-disk – a pseudo disk that uses main memory, as its backing device. tmpfs serves as a metric for the overheads involved in traditional file-system-based data movements, even in the absence of costly crash-consistency semantics. We store each of the variable states in a new file at the program checkpoint where the versioning information is encoded into the file name.

pmfs: is PMEM aware file system that supports crash-consistent data storage. `pmfs` [8] internally treats PMEM as a memory device underneath its POSIX file-system interface, thereby eliminating the page-cache and device driver layers. We emulate an PMEM device for PMFS by reserving a portion of DRAM at the OS start-up [37]. We use the PMFS implementation [38] ported for a newer Linux 4.x based kernel. Program snapshots are encoded into files similarly to `tmpfs`.

nocheckpoint: represents the application execution time without any checkpoint-related data movements representing the best case execution time.

3.6.2 Data Movement Latency

First, we evaluate `NVStream` against a home-brewed microbenchmark named NVSB. NVSB emulates an I/O dominant MPI-based compute kernel that periodically checkpoints its matrices after every configured number of iterations. We use NVSB to evaluate the sensitivity of each I/O strategy towards data movement granularity. To this end, we vary the variable sizes used in NVSB in each run, while keeping the total checkpoint data size constant. We run a single-threaded (one rank) benchmark instance with 400MB iteration checkpoint data size and plot the results in §5.

`NVStream` outperforms file system-based I/O for all data granularities. The `NVStream` I/O is 2.6× faster compared to `tmpfs` and as much as 31× faster than `pmfs` I/O at 32KB

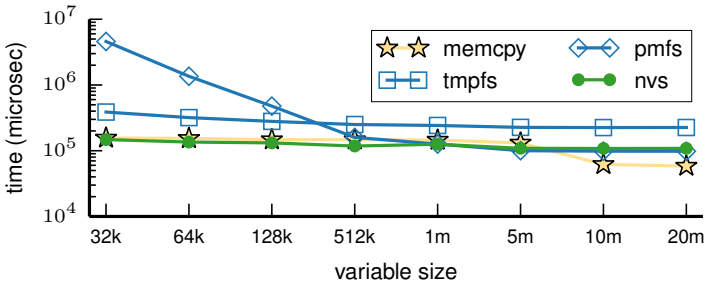

Figure 5: Checkpoint/analytics data write time against different storage stack transports, namely tmpfs, memcpy, pmfs and nvs. y-axis uses log-scale

variable size. It is important to note that `pmfs` performs as well as `NVStream` for large variable sizes (5M, 10M). However the I/O performance decreases exponentially as the variable sizes get smaller. We attribute this behavior to the crash-consistency overhead of `pmfs`. Smaller variable sizes increases the number of new file creation, therefore increase the file-system metadata manipulations. Metadata updates involve executing costly undo-logging based crash-consistency routines thus lowering the I/O throughput of `pmfs` at small I/O granularity. `NVStream` I/O performs equally well at every I/O data granularity. This is because `NVStream` uses data batching during checkpoints, effectively converting multiple I/O calls into one large I/O operation. `NVStream` performs as well as, or even slightly better than `memcpy` based I/O. This is because a `memcpy` call for every small variable prevents memcpy from utilizing the full memory bus bandwidth due to serialization. However for larger I/O sizes (e.g., 10MB), `memcpy` outperforms `NVStream` I/O by as much as $2\times$.

Next, we evaluate `NVStream` with the three HPC applications, GTC, CM1, and mini-AMR. We configure the GTC test setup such that each rank outputs $\sim 210MB$ of I/O data per iteration. We run the benchmark for increasing number of MPI ranks(N value) while keeping the per rank output data size more or less the same (weak scaling). We report in Figure 6 the average data I/O time for each I/O mechanism, normalized to `memcpy` I/O time. `NVStream` I/O is 24% faster than `pmfs` when 'N=4', whereas `pmfs` I/O performance drops drastically with the number of MPI ranks. As a result at 'N=64' `NVStream` I/O is $10\times$ faster than `pmfs`. NVSD moves $\sim$ 50% less I/O data compared to other I/O techniques and is 33% faster compared to `NVStream` I/O and `memcpy`.

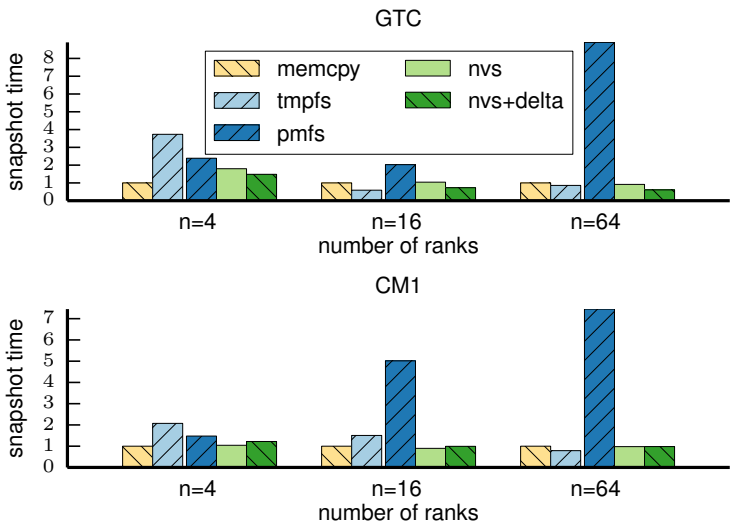

Figure 6: GTC and CM1 snapshot time for each of the I/O techniques. We normalize the times to best case data movement time – memcpy. We run each benchmark with increasing number of MPI ranks.

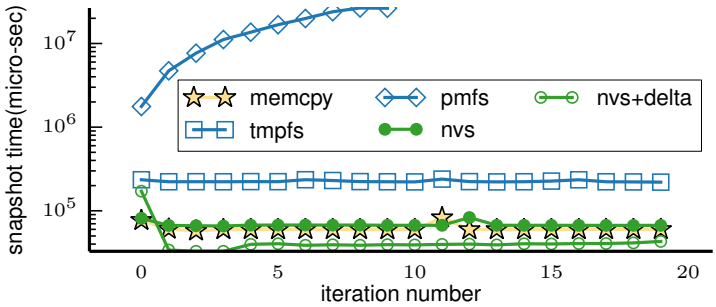

Figure 7: Program snapshot time of miniAMR plotted against checkpoint iteration number. y-axis uses log-scale

Similarly, we configure the CM1 application to output $\sim$ 45MB of I/O data per-iteration/per-rank. Unlike GTC, CM1 is a compute heavy HPC kernel, thus increasing the I/O data size leading to more time spent in the compute portions of the benchmark. We report in Figure 6 the average data I/O time for each I/O mechanism normalized to `memcpy` I/O time. `NVStream` I/O is 7$\times$ faster than `pmfs` and NVSD I/O performs the same. It is important to note that the performance of `NVStream` I/O is as fast as `memcpy` based I/O in most occasions, which confirms the lightweight crash-consistency semantics of `NVStream`.

miniAMR [39] does not have an inbuilt checkpoint routine, and therefore, we implement it by saving the current mesh/blocks after each compute iteration. We use the application supplied sample workload *Two moving spheres* as our benchmark run. However the

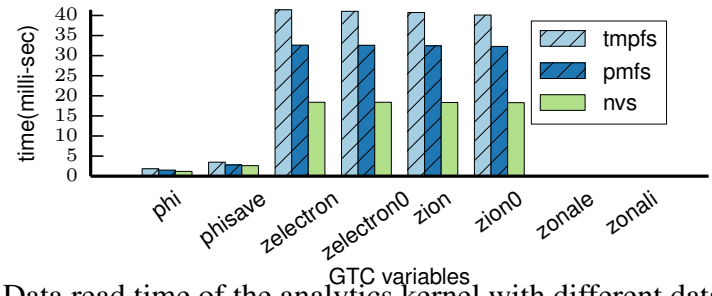

Figure 8: Data read time of the analytics kernel with different data transports

default max block size of 4000 results in a large number of variables and drives our `pmfs` setup to a non-responding state. We use 400 as the max number of blocks parameter and run the application with 4 ranks (see Figure 7). The `pmfs` based file I/O is 500× slower compared to `memcpy` at its worst-case data point. `NVStream` I/O performs within 11% margin of `memcpy` – the best case I/O mechanism, and NVSD outperforms the best case by 27%.

Next, we couple the GTC simulation output with a data compression analytics kernel [40] and evaluate the `NVStream` data read performance. Each rank of the analytics kernel reads the GTC variables in a monotonically increasing version order – the most common data read pattern found in the coupling workflows. We record the time spent on data reads in the analytics kernel under each of the I/O mechanisms and report the results (see Figure 8) normalized to `memcpy` based I/O reads. The analytics kernel runs with 16 MPI ranks where each rank consumes a GTC application output belonging to a single rank, thereby creating a one-to-one coupling testbed. We observe that `NVStream` reads are 43% faster compared to `pmfs`.

3.6.3 Delta-compression

Next we evaluate the ability of NVSD– the delta compression enabled `NVStream`– to reduce the total program snapshot size over simulation runs. The data size reduction will directly contribute to reduction in system interconnect bandwidth usage during data movements, data read time at the analytics application endpoints, and data movement across the compute nodes. In these experiments, we record the total program checkpointing size of each application with and without delta compression, shown in Figure 9.

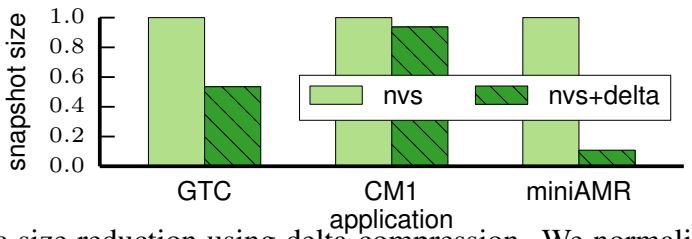

Figure 9: I/O data size reduction using delta-compression. We normalize the data size to `nvs`

In both GTC and CM1, we observe that the data access pattern across the iterations stays the same, keeping the output data amount at a fairly constant level across iterations and ranks. We report the average I/O reduction across compute ranks. NVSD reduces GTC program snapshots by as much as 47%. The biggest data reduction contributions are from variables that get conditionally activated during execution. The original GTC code selectively checkpoints objects based on the static configuration options, thus the checkpoint routine has complex conditional structures. `NVStream` eliminates the need for this application specific logic from the checkpoint routines, and delivers the same or better level of checkpoint performance without developer intervention. In contrast, CM1 allocates around $\sim$80 variables per MPI rank, with a high modification factor among most of the variables. Therefore, NVSD yields only a marginal data reduction of 7% over `NVStream`.

miniAMR [39] differs from both GTC and CM1 as it refines the initial mesh during simulation (e.g., sphere moving through 3-dimensional mesh). The visual inspection of the application code reveals that it allocates its global mesh data structures during the application initialization, similarly to most other HPC applications. Our initial memory profiling on the application shows that the modification factor among large variables is very low. Our experiments further confirms our hypothesis. As a result, NVSD yields in data size reduction as much as 99% in some ranks.

3.6.4 Impact on Simulation Time

The ability of NVSD to compress output data comes at the cost of page-protecting and signal handling, which introduces additional compute overheads. To quantify this, we measure

the effects of NVSD on application execution time. We highlight NVSD in this section because it alters the compute kernel's running time and tracks the data updates during the compute operation. The remaining I/O mechanisms, including `NVStream`, only get activated during the data write/read routines of the application. We measure the iteration time of each of the application and report the execution overhead against `nocheckpoint` I/O representing the best case execution time of the application.

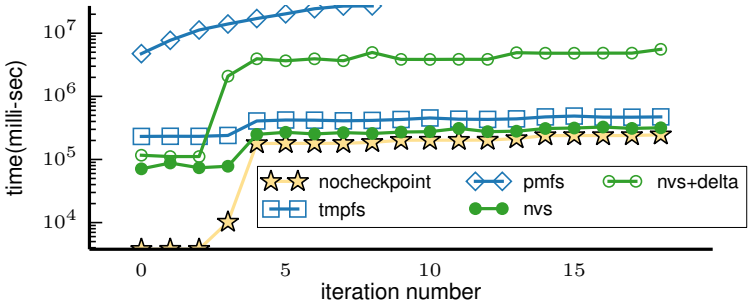

Figure 10: Iteration time cross section of the miniAMR against different snapshot schemes. Please note that y-axis uses log-scale

The bandwidth of our emulated PMEM device is same as the DRAM bandwidth of our testbed as we do not throttle the bandwidth of the emulated device. In addition, the compute requirements of GTC and CM1 are such that for smaller evaluation scale of the experiments, the data movement overhead of all PMEM-based I/O mechanisms remains small in comparison to their compute phase. The `pmfs` based program snapshots incur 2% of execution overhead for GTC at 'N=64'. `NVStream` I/O contributes to similar overhead and NVSD increases the iteration time of GTC and CM1 by 2% and 0.5% respectively.

In contrast, the checkpoint routines of miniAMR are executed frequently enough to make the I/O overhead significant enough at the program scale used in our evaluation. With miniAMR, the `pmfs` based file I/O costs increase drastically (see Figure 10), incurring iteration overheads as much as 66× over `nocheckpoint` time. This is because, as we highlighted in §3.6.2, `pmfs` suffers high performance degradation due to frequent meta-data updates. However, `NVStream` checkpoints only incurs 47% overhead over `nocheckpoint` iteration time. Finally, NVSD increases the iteration miniAMR iteration time by as much

as 20× at the benefit of significant reduction in checkpoint size as shown in §9.

3.7 Chapter summary

In this chapter, we explore the ways of accelerating node local HPC I/O using new persistent memory hardware, while keeping the changes to the original block based persistent I/O APIs, minimal. We discuss `NVStream`, a PMEM aware persistent I/O stack supporting a memory friendly object API. The memory APIs of `NVStream` support meta-data tagging including object versioning and are flexible enough to emulate block I/O based checkpointing routines in legacy HPC applications. `NVStream` avoids costly overheads of OS I/O subsystem by implementing these APIs as a userspace library runtime. Furthermore, `NVStream` leverages byte-addressability of PMEM, streaming memory writes in modern CPUs and insights on application workloads to design and implement a fast storage engine with optimized durability semantics. `NVStream` also supports delta compression on I/O data, which further reduces I/O cost for workflows with higher write locality. The evaluation of `NVStream` using I/O benchmarks and scientific applications demonstrates 10× reduction in I/O compared to PMEM-optimized file systems.

CHAPTER IV

BANDWIDTH AWARE I/O WITH PERSISTENT MEMORY

The previous chapter focuses on designing a PMEM optimized I/O stack for existing block I/O applications. The work enables low latency I/O data movement from applications onto PMEM storage. However, application I/O can still suffer from the limited bandwidth of the data movement channel among applications and the PMEM storage media. The limitation becomes even more prominent when application data movement volumes are large - a common use case in HPC applications. In this chapter we discuss a system software technique to overcome this problem.

4.1 Introduction

HPC applications frequently output their application state (local checkpoints) on to node local durable media. Fast local checkpointing is important for accelerating in-situ analytics pipelines [41] that consume the checkpoints for further scientific insight generation, and multi-level checkpoint-restart (C/R) facilities [42] that uses local checkpoint state to recover from frequent soft failures in HPC engines. PMEMs integrated in future exascale HPC engines [43] is an ideal storage media for these use-cases as they provide both low latency read/writes and high capacity storage. However, current HPC system software stacks supporting I/O cannot fully exploit the technological advantages of PMEM resulting in an inefficient PMEM use and significantly lower performance benefits.

For use of PMEM in C/R, prior work [44, 45, 46] has proposed methods to incorporate newer and faster storage technologies like Flash/SSD and PMEMs as stable storage. These studies have shown that Flash-based storage is not sufficient to address the increase in per-node core count and the data-to-core ratio. Recent work on C/R uses byte-addressable PMEM as stable storage that can significantly improve storage performance compared to

block-based SSDs [47, 48]. Although PMEMs are expected to deliver 100x faster access compared to SSDs, current PMEM technologies are limited by (i) relatively low per-node socket bandwidth (1-2 GB/sec die bandwidth compared to >10 GB/sec for DRAM), and (ii) high write latency (3-5x slower than DRAM). Concerning large volumes of checkpoint data transfer across DRAM and PMEM (a norm in HPC applications), and high core count (64-128 cores/nodes in exascale) will severely limit the benefits of PMEM.

Chapter §3 of this thesis, introduces `NVStream`– a PMEM aware userspace I/O library runtime. `NVStream`'s memory friendly I/O APIs and PMEM optimized storage engine remove critical path data movement overheads of HPC I/O using a userspace I/O stacks, PMEM and application friendly data layouts and optimized crash-consistency protocol. However, `NVStream` and similar prior work [48, 49] are limited by the properties of the PMEM technology as the checkpoint I/O time is only governed by the device bandwidth.

In this chapter we present Phoenix (`PHX`) – an PMEM-bandwidth aware library for checkpoint I/O, as the key technical contribution. Similar to `NVStream`, `PHX` uses memory-centric interfaces supporting object versioning, but solves the limited PMEM bandwidth through simultaneous use of PMEM and local/ peer nodes' DRAM devices, thus increasing the effective data movement bandwidth. `PHX`'s memory-centric object interface and PMEM-bandwidth-aware design lead to reduction in the time length of I/O operations in the critical path, associated with the slow PMEM device. To continue guaranteeing adequate reliability and persistence, DRAM-resident object state is replicated across peer nodes' memory, which is accessible through high-bandwidth interconnects. `PHX` is implemented and evaluated with several representative HPC applications – 3D Gyrokinetic Toroidal Code (GTC), CM1 and S3D

4.2 PMEM Limited Bandwidth

Recent work [50] identifies the slow reads/writes of PMEM as a major limitation during computational memory accesses (e.g., PMEM as the main memory for HPC applications).

However, unlike the use of PMEM as computational memory, where frequent but fine-grained data accesses are the norm, HPC I/O results in occasional but large data movements in/out of the storage medium. We attribute the aforementioned HPC I/O behavior to the iterative nature of the HPC applications, where analytics or checkpoint data is output after a certain number of compute iterations. Thus, when used solely as an HPC I/O storage medium, it is likely that the limited device bandwidth of the PMEM will become the major bottleneck during I/O data movements. Therefore in this work, we mainly focus on the limited device bandwidth of the PMEM. To that end, we identify two broad classes of techniques that minimize the time spent during I/O data transfer with PMEM.

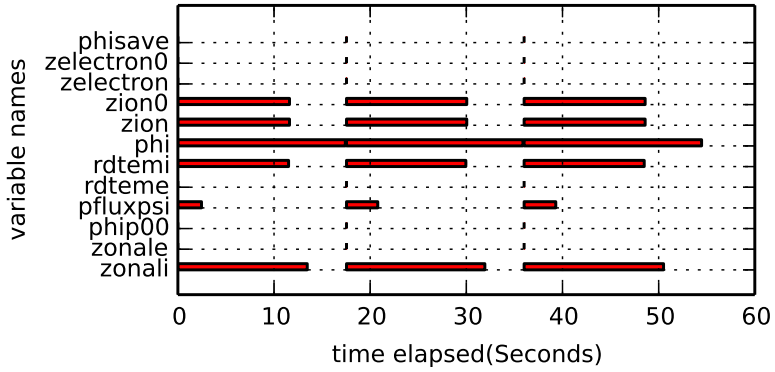

Figure 11: Access patterns of GTC variables between checkpoint iterations. Apart from variable 'phi' all the other variables are viable candidates for pre-copy.

Reduce the critical path bound I/O data. Speculative data movement is a well-known mechanism to alleviate device bandwidth limitations. The technique, pre-copy the I/O data into target device, thus shifting some of the data movement costs out of the critical path of the application execution. Similarly, incremental checkpoint techniques avoid redundant writes and hence save the precious device bandwidth. Figure 11 shows a trace map of GTC showing the last access time of each of the checkpoint variables. The trace map shows that many of the variables (e.g., phisave, zelectron) are last accessed at the very beginning of the iteration, and few others are accessed at the halfway point between checkpoints, making all of them viable candidates for pre-copying. However, speculative I/O and incremental

checkpoints heavily depend on the application's variable access patterns. Some applications (e.g., CM1 and S3D) may modify checkpoint variables up until the checkpoint time, and often enough, thus denying either of these optimizations.

Use of alternative interconnects in the system. Today's systems are not isolated computing resources. With deep memory hierarchies and fast inter-node communication links such as InfiniBand, there is more than one way to move data into storage mediums. For an instance, the proposed Summit [51] machine is equipped with 23GBps point to point interconnects and the number is expected to go up to 50GBps [52] in the near future. Coupled with RDMA, they put remote DRAM memory on par with PMEM in terms of per-core bandwidth scaling. Figure 12 shows the data movement times, plotted against the fraction of data that gets written to remote storage over the interconnect. The two plots correspond to different bandwidth ratios between local and remote storage. It is clear that use of the *aggregate-bandwidth* of both local and remote storage lead to the best data movement performance.

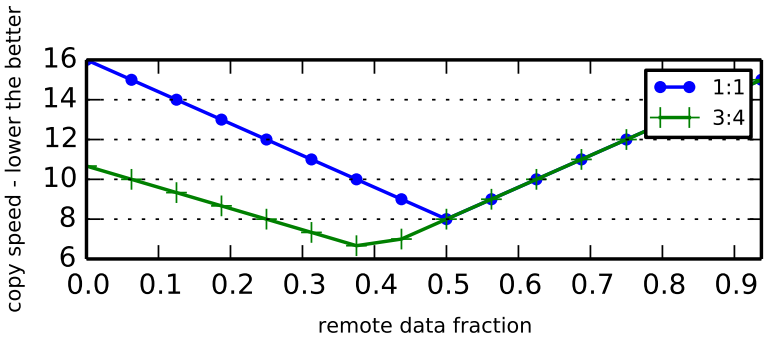

Figure 12: Data transfer time plot against the fraction of data being copied over interconnect. The use of aggregate bandwidth gives us faster I/O times. The legend represents the bandwidth ratio between the two storage devices

4.3 Phoenix Overview

`PHX` presents the concept of *persistent objects*. All types of HPC I/O are modeled as operations on memory objects accessed via memory-based APIs. Applications declare the

expected object characteristics (persistence, reliability, versioning) during the initial object allocation. At the heart of the Phoenix is an object tracking runtime that provides object versioning, reliability, and persistence.

The base APIs of Phoenix runtime are as follows:

- **init()** - initialize the Phoenix library runtime.
- **create_obj(key, size, unit_size, properties)** - allocate an object from main memory and sets its property (persistence, versioning, etc) details; return a the memory pointer to the caller.
- **destroy_obj(key)** - de-allocate the memory space of a given object key.
- **finalize()** - clean-up library resources.

PHX also introduces a checkpoint-specific operation 'checkpoint_commit()' that moves data residing in volatile memory into the stable storage (PMEM). During the memory allocation programmer explicitly specify checkpoint objects via the property, 'CHECKPOINT=true', and by default, 'checkpoint_commit()' operate on all such checkpoint objects. Figure 13 shows a simple template for implementing coordinated checkpoints with PHX APIs. The same object APIs can be used (see Figure 14) to load previously checkpointed data into newly allocated memory with 'LOAD_DATA=true' property with appropriate version details.

4.4 Design and Implementation

4.4.1 Hiding PMEM Read/write Latencies

Most of the current HPC applications use file-system APIs to read/write data from their stable storage medium. This is because current HPC engines use block-based devices such as disk arrays and SSDs as the underlying stable storage. However, PMEMs are byte addressable storage which motivates us to treat them as a slow memory instead and also

```
char key[]="foo";
/*create checkpointable object instance*/
struct properties prop = {.checkpoint = true, ...};
void *ptr=create_obj(key,SIZE,prop);
.
/*do computation using data_ptr*/
.
checkpoint_commit();
```

Figure 13: Using PHX-C/R to carry out coordinated checkpoints

```
char key[]="foo";
/*This is a restart run. Load allocations with
 *   checkpoint data */
struct properties prop = {.checkpoint = true,.restart=true, ...};
void *ptr=create_obj(key,SIZE,prop);
.
/* resume computation */
.
```

Figure 14: How to load previously checkpointed data in to object allocations, during a restart run

use the same virtual memory APIs to access them. Treating PMEM as memory allows PHX (1) to operate on page level granularity while moving data in/out, and (2) to make use of hardware support for memory read/write protection on data portions on PMEM. HPC workloads are long running applications with significant memory accesses. Hence, due to high PMEM write latencies, running applications directly (execute-in-place) from PMEM is not feasible. As a result, we allocate application object from DRAM as usual and move data in/out from PMEM storage structures during reads/writes.

The use of a memory like programming abstraction over PMEM allow system software developers to use proven functionalities of OS VM subsystem and enable fast I/O by eliminating application to kernel crossings of file I/O by using specialized data serialization on the PMEM.

4.4.2 Overcoming the Limited PMEM Bandwidth

PMEM devices are expected to have 4-8× lower per-channel bandwidth compared to DRAM, and as explained in §4.2, it is likely to become the most bottlenecked resource during I/O data movements. `PHX` uses DRAM bandwidth to alleviate the limited PMEM bandwidth during data output. The key idea is to use aggregate device bandwidth of both DRAM and PMEM during I/O data movement, thus shortening the critical path data movement time. Towards that end, first Phoenix splits the total critical path I/O data into two parts: DRAM-bound data (Ddata) and PMEM-bound data (Pdata), while taking device bandwidth ratios and DRAM capacity budget into consideration. During a bulk I/O data movement, `PHX` moves a copy of Ddata into a DRAM buffer, and in parallel moves Pdata over to PMEM (see Figure 15). However, the DRAM is volatile and hence saved Ddata has to be quickly moved from DRAM to PMEM and committed – a critical property for achieving the data persistent property for written I/O data. We refer to this operation as `I/O de-staging`, and the DRAM-resident temporary buffers as `staging buffers`.

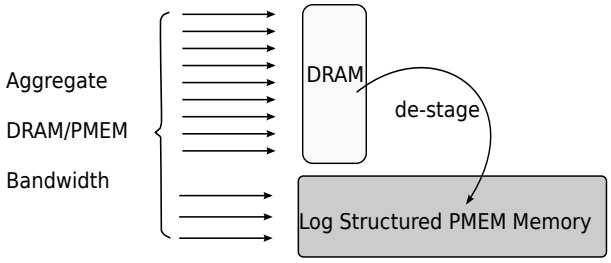

Figure 15: I/O data gets written to both local PMEM and DRAM to exploit aggregate bandwidth of the devices. Data de-staging from DRAM to PMEM ensures the bounded DRAM memory usage.

Next, moving all of the output data to persistent storage, involves de-staging of Ddata on to the PMEM, where data movement costs are overlapped with application execution. With the presence of many (including cheap) cores on current and future exascale platforms, the de-staging can be performed using dedicated CPUs in the background, and the

commensurate gain compensates the CPU time used for de-staging in the application execution time.

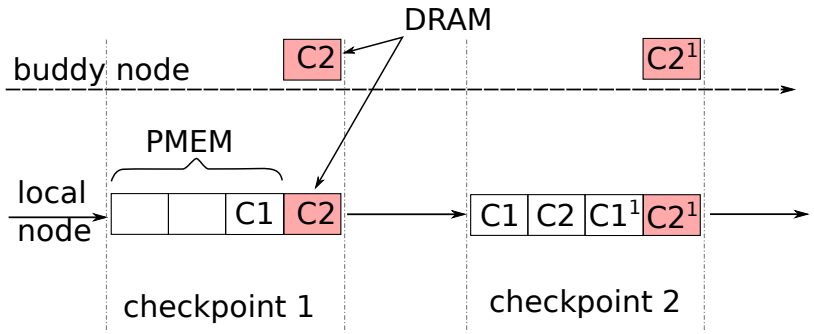

Figure 16: Contents of the PMEM/DRAM, as the checkpointing progresses. Total checkpoint data is split up and written to both DRAM (staging buffer) and PMEM checkpoint log

However, the Ddata de-staging phase is vulnerable to soft/hard node failures, as such failure would wipe out the staged data in the DRAM buffers. `PHX` recognizes the fact that not all HPC output operations expect the same levels of data persistence during their I/O and exploit it for I/O acceleration. `PHX` also provides strict output data persistence guarantees and we describe the details in the next section.

Poor bandwidth scaling of PMEM will become the bottleneck during bulk data output, which is the norm in HPC I/O. `PHX` overcomes the limited PMEM bandwidth by using much superior DRAM bandwidth. Volatility and poor capacity scaling of DRAM deny us from using it as the sole storage device. Thus we use DRAM and PMEM's aggregate bandwidth for fast data movement and also ensure that the data in DRAM is made persistent by de-staging.

4.4.3 Reliability requirements of HPC I/O

The naive use of the aggregate bandwidth approach falls short of providing the required data reliability guarantees for I/O, because part of the output data remain in DRAM before being de-staged to PMEM, making it vulnerable to data loss in the event of node

failure. However, in current supercomputers [53], the network interconnects offer point to point bandwidths up to 56 Gbps and these numbers expected to be 100-400Gb/s [52] in future HPC machines. The fast interconnect bandwidth and remote direct memory access (RDMA) stacks make the cost of writing/reading remote DRAM memory cheaper than that of with local PMEM memory. Hence, it is viable to achieve local DRAM fault tolerance by using remote replication and achieve data transfer times on par or faster than local PMEMs. Replicating data to N nodes, i.e., an N-node replication scheme, will enable recovery from up to N-1 simultaneous node failures.

Phoenix uses N=2 replication scheme for its DRAM staged data (Ddata). For each of compute node, we assign a buddy node to act as the remote DRAM node. During the aggregate bandwidth copy, we write Ddata to both local DRAM and to the buddy node's DRAM (using RDMA). Figure 16 shows the checkpoint data placement in PMEM, DRAM and the buddy's DRAM under an N=2 replication scheme. §4.4.6 shows that an N=2 replication scheme would bring down the failure probability of staged data to a significantly lower level, even at the projected exascale failure rates.

In this manner, *Phoenix enables applications to use the aggregate bandwidth of both DRAM and PMEM while meeting the necessary reliability guarantees of application data. To that end, PHX creates a software defined storage layer with sufficient reliability guarantees by combining two storage mediums with heterogeneous data persistence guarantees.*

4.4.4 Move Data Copy Out of Critical Path

Data pre-copying is a well-known mechanism used in HPC I/O where we speculatively copy the data into output buffers/storage, thus reducing the critical path data movement time at the I/O commit. Recent studies have explored the technique, in the context of PMEM based HPC I/O as well [49, 54]. PHX complements its aggregate bandwidth based checkpointing mechanism with data pre-copying to further improve the C/R performance.

PHX learns the application data access patterns, during the initial checkpoint iterations

and records the checkpoint object's last access time since the previous checkpoint (referred as time-offset from here onward). In subsequent iterations, PHX predicts the time-offset based on its learning phase and starts speculatively copying the I/O data into PMEM at the end of predicted time-offset. However, PHX may mispredict the time-offset, as the processor power levels, OS scheduling, etc., affects the application execution time between checkpoint iterations. Miss predictions fall into two groups: (i) the predicted time-offset is earlier than the actual time-offset thus we have pre-copied stale data, or (ii) the predicted time-offset has a significant overshoot value from the actual value, making the pre-copying less efficient. We write-protect the application object memory locations prior to pre-copying (a similar mechanism used in [49]), to detect the former miss-prediction type. Write protection allows detection of changes to the original application object after it has been pre-copied to the output buffers/storage. We adjust the pre-copy time offset dynamically during the subsequent iterations using a technique similar to TCP bandwidth search.

We correct the predicted time-offset due to type-(i) miss-predictions by increasing the value (adding $\delta.t$), and type-(ii) by decreasing the same (subtracting $\delta.t$), but only after M(configured) number of iterations. It is important to note that the pre-copy mechanism does not affect the correctness of the output data, that is miss predicting a pre-copy only results in wasted bandwidth during the early copy. In summary, *application access patterns may allow pre-copy optimization on the output I/O and provide opportunity to minimize the PMEM data copying in the critical path.* PHX *implements a pre-copy mechanism that augments the effective aggregate bandwidth for C/R. Some applications are immune to this optimization because of their late data access behaviour.*

4.4.5 Phoenix Heap Manager

PHX identifies two types of memory resident data and allocations. First, there are DRAM resident application memory objects for in-place execution. The second type are memory locations for storing checkpoint state in the form of local DRAM buffers, local PMEM

and remote DRAM allocations (storage memory). Applications make the object allocation requests for the first type of memory. The APIs (see Figure 13) are similar to a traditional user space memory allocation, and only differ from the input metadata it permits. The PHX object allocator allows programmers to tag requested memory with additional properties such as object handle, versioning, and persistence. The given properties decide, how the memory resident objects should be handled by each of the PHX services and core runtime during application I/O.

Next, PHX manages memory used for I/O data storage (storage allocator). PHX uses local DRAM, local PMEM and remote DRAM for placing persistent I/O data (checkpoints) during aggregate bandwidth checkpoints, thus the storage allocator allocates and manages memory from all of the above physical memory locations. Local DRAM and remote DRAM allocated memory are significantly smaller (and bounded by the explicit memory budget) than local PMEM allocation as they are only used for temporary I/O staging during the aggregate bandwidth I/O. PHX stores the I/O data over the allocated storage memory by serializing the I/O data into log structured memory. PHX maintains a meta-data index of log-structured data in a separate ring buffer structure at a well known PMEM storage location (start of the PMEM storage memory). The meta-data structure only occupies 64 B per object, where HPC objects on average occupy hundreds of thousands of megabytes, therefore by separating the metadata from actual data we enable fast cache friendly object lookups and traversals and less lock contention during data copy. Each of the meta-data entry records, object handle, version, pointers to their actual data (may point to local PMEM, local DRAM, or remote DRAM).

4.4.6 Failure Analysis of Staged I/O

Next, we consider the failure probability during I/O data staging and evaluate the effectiveness of our application run ahead strategy. Our calculations are based on a simple model similar to [55]. Figure 17 shows the de-staging and the application run ahead mechanism

of PHX.

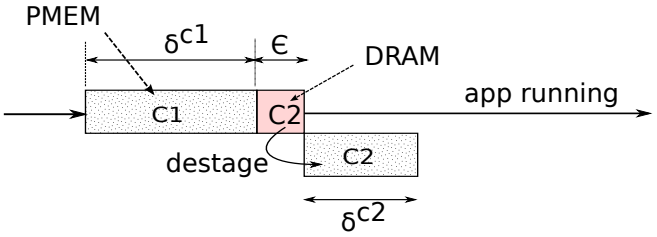

Figure 17: Application run ahead during de-stage time

Let the number of processors be n, and the failure rate of each processor be λ. if M is the MTBF of each processor then $\lambda = 1/M$. For an HPC application with failure free execution time of R, the failure probability of each processor is $\lambda.R$ and the total system failure probability is given by $1-(1-\lambda.R)^n$.

For Phoenix, we define an upper bound t on the de-staging time after each checkpoint iteration.

For a stage buffer replication scheme of N=1, that is a stage buffer is only maintained in local DRAM, the probability of Phoenix loosing the staged data (i.e., unrecoverable failure) at any given processor system-wide, during the de-staging time interval t is given by $1-(1-\lambda.t)^n$.

A stage buffer replication scheme with N=2 would consist of $n/2$ number of buddy groups and therefore PHX aggregate bandwidth checkpoint fails only if both replicas fail simultaneously. The probability of unrecoverable error in a processor, during time interval t, given that its buddy has already failed, would be $(\lambda.t)(\lambda.t)$. The system wide unrecoverable failure probability p of the scheme (N=2 replicas) would be $1-(1-\lambda^2.t^2)^{n/2}$. Therefore the failure probability of m consecutive Phoenix checkpoints would be given by $1-(1-p)^m$.

Table 3 lists down the failure probabilities, calculated using our failure model. We show the failure probabilities under different replication schemes and application execution times. As per the calculations, a no-checkpoint application with 24 hour running time

Table 3: Unrecoverable failure probabilities of Phoenix C/R compared with a free run. MTBF of each processor = 20 yrs, processor count n = 100,000, and m = 2880 (~48 hrs/60 s checkpoint iteration)

Scheme	**Execution time**	**Checkpoint interval**	**Max de-stage interval**	**Failure probability**
Free run	24 hrs	-	-	0.99999887
PHX, N=1	48 hrs	60 s	10 s	0.98960270
PHX, N=2	48 hrs	60 s	10 s	0.00000003

Table 4: component values used in energy analytical model. Lowercase 'e' denotes the unit energy cost.

e_{pmem}	PMEM data write
$E_{aggregate.bw}$	aggregate bandwidth checkpoints
e_{dram}	DRAM data write
$e_{remote.dram}$	remote DRAM data write
$e_{pmem.destage}$	pmem data write during de-stage
$E_{data.mv.overhead}$	additional data movement during aggregate bandwidth checkpoints
$e_{interconnect}$	data movement via network interconnect
$e_{switching}$	data switching
e_{link}	data transfer in network links
$e_{interface}$	data transfer via network interface
E_{saved}	saved energy due to fast checkpoints
$E_{core.energy}$	energy budget per core

has very high (if not certain) failure chance. The N=2 replication scheme of PHX would bring down the unrecoverable failure probability of staged data to 3.2×10^{-8} for an extended running time of 48 hours. That is, PHX can recover from all system wide soft-errors for a period of 48 hours, with a probability of 0.99999997. An unrecoverable staged data failure will force PHX to recover from its last PFS based remote checkpoint, thus a careful selection of remote checkpoint frequency (48 hours in the above calculation is too pessimistic) will enable PHX to keep staged data failures at a negligible level. In summary, *PHX's DRAM staged data, is vulnerable to data loss during node failures. With careful selection of replication scheme and remote checkpoint frequencies, it is possible to bring down the staged data loss probability to negligible/acceptable levels.*

4.4.7 Energy Model

While PHX C/R speeds up the checkpoint times, it incurs additional energy overheads compared to naive PMEM checkpoints, due to extra data movements (de-staging/ buddy checkpoints). Moving towards exascale computing, the HPC community puts a lot of emphasis on energy efficient computing as they try to keep the operating energy budget of future exascale machines under 20MW [56]. Thus it is important to understand the energy requirements of proposed PHX C/Rs. In this section we build an abstract model to analyze PHX's energy overheads/ savings. Table 4 describes the notations used, in our energy model.

The total energy consumed by naive PMEM based checkpoint, for a data of size C given by, $C.e_{pmem}$. The total energy consumed by aggregate bandwidth checkpoints (with N=2 replication scheme) for the same data size, given a fraction P of the checkpoint data, being Ddata (staged) is given by,

$$E_{aggregate.bw} = C.(1-P).e_{pmem} + C.P.e_{dram} + C.P.e_{remote.dram} + C.P.e_{pmem.destage} \quad (1)$$

We approximate,

$(1-P).e_{pmem} + P.e_{pmem.destage} = e_{pmem}$

thus the energy overhead due to additional data movements of the aggregate bandwidth checkpointing is given by,

$$E_{data.mv.overhead} = C.P.e_{dram} + C.P.e_{remote.dram} \quad (2)$$

Moreover, we express the $e_{remote.dram}$ in terms of interconnect energy and DRAM write energy cost e_{dram}.

$$e_{remote.dram} = e_{interconnect} + e_{dram} \quad (3)$$

Hence the additional data movement energy cost of PHX aggregate bandwidth checkpoints (with N=2 replication) over the naive PMEM checkpoints, is given by,

$$E_{data.mv.overhead} = C.2P.e_{dram} + C.P.e_{interconnect} \tag{4}$$

and,

$$e_{interconnect} = e_{interface} + e_{switching} + e_{links} \tag{5}$$

Similar to [57] we assume the interface energy cost to be negligible compared to other energy components.

Our proposed checkpoint mechanism incurs energy overheads due to additional data movements. However, it reduces the checkpoint time while doing so, and the total simulation time, thus resulting in energy savings from the simulation execution. If the reduction of checkpoint time is t, the saved unit energy value is given by,

$$E_{saved} = e_{core.energy}.t \tag{6}$$

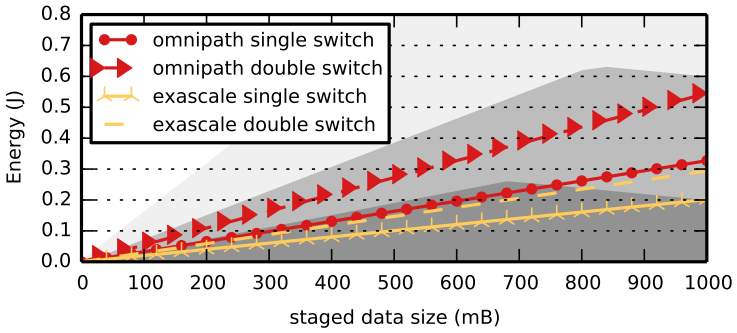

Figure 18: Curved plots (dark, medium and light shaded areas) show the energy savings of PHX C/R over naive PMEM C/R, under different DRAM : PMEM bandwidth ratios, 2:1, 4:1 and 8:1 respectively. line plots refer to Omni-Path [58, 57] and expected exascale interconnect [57] energy costs, during N=2 replication scheme, while the buddy node distance being single/ two switches away.

Table 5: Energy values used for each of the technology component. [57]

checkpoint size	1024 MB
energy budget per core	0.2 J/s
Omni-Path energy	21 pJ/bit
exascale switch energy	6 pJ/bit
link energy	5 pJ/bit
DRAM write energy	4 pJ/bit

Figure 18 plots, both saved and extra energy overheads of PHX C/R, due to additional data movements, using the derived energy model, for a total checkpoint workload of 1024 MB per core. Table 5 lists the energy costs we used for our calculations. We plot the saved energy under varying DRAM to PMEM bandwidth ratios and, it is clear that energy savings are increased as we increase the bandwidth ratios. Additionally, the energy saving curves have a peak energy reduction point, and it corresponds to the most efficient data split between DRAM and PMEM. The straight lines represent the data movement energy costs, and it is a linear function of the remotely copied data. Furthermore, the data movement cost also depends on the network proximity (how far is the buddy node) and the switching/interconnect technology. For an example, if the DRAM to PMEM bandwidth ratio is 2:1, and the buddy node is one switch away (Omni-Path interconnect), PHX can stage checkpoint data up to ∼750 MB without paying additional energy costs. In summary, based on this analytical model *the PHX C/R results in extra energy overheads, due to additional data movements. However it also results in energy savings, in the form of total application execution time reduction. Thus with careful selection of PHX configuration and runtime parameters it is possible to obtain fast checkpoints with energy savings, or at no extra energy costs.* Our future work will further explore opportunities for experimental evaluation of the energy-efficiency related impacts of PHX.

4.4.8 Checkpoint Model

The PHX checkpoint service uses aggregate bandwidth I/O to speed up the local checkpoints (more frequent) and uses remote PFS for global checkpoints (less frequent), thus creating a multi-level checkpoint scheme. In our effort to derive an abstract model for PHX-C/R we

Table 6: Notations used in C/R model

T_S	original computation time of a workload
p_L	percentage of local checkpoints
p_G	$1-p_L$, the percentage of global checkpoints
τ	local checkpoint interval
δ_L	local checkpoint time
δ_G	global checkpoint time
δ_{eq}	checkpoint overhead in general
R_L	local checkpoint recovery time
R_G	global checkpoint recovery time
R_{eq}	checkpoint recovery time in general
q_L	percentage of failure covered by local checkpoints
q_G	$1-q_L$, the percentage of failure that have to be covered by global checkpoints
MTTF	system mean time to failure, modeled as 5 year/number of nodes
T_{total}	total execution time including all the overhead

identify that our local/global checkpoint strategy is similar to Dong et al.'s [48] model of C/R. Hence we extend their model with our `PHX` checkpoint parameters. Dong et al. derive (refer Table 6 for notations) the execution time of hybrid checkpoint, starting with,

$$T_{total} = T_S + T_{dump} + T_{rollback,recovery} + T_{extra-rollback}$$

they derive,

$$T_{total} = T_S + \frac{T_S}{\tau}(\delta_{eq}) + (\frac{1}{2}(\tau + \delta_{eq}) + R_{eq})\frac{T_{total}}{MTTF} + \frac{p_L q_G}{2p_G}(\tau + \delta_L)\frac{T_{total}}{MTTF}$$

where,

$$\delta_{eq} = \delta_L.p_L + \delta_G.p_G \tag{7}$$

$$R_{eq} = R_L.q_L + R_G.q_G \tag{8}$$

`PHX` checkpoints differs from their model by defining local checkpoint time in terms of local PMEM checkpoint time of Pdata (δ_{LN}), local DRAM checkpoint time of Ddata (δ_{LD}) and remote DRAM (buddy) checkpoint time of Ddata (δ_{RD});

$$\delta_L = Max[\delta_{LN},\ \delta_{LD},\ \delta_{RD}] \tag{9}$$

For a given checkpoint size C, and $C = C1 + C2$, where C1 and C2 are the Pdata and Ddata checkpoint sizes, respectively, the time to complete the checkpoint under Phoenix with N=2 replication would be,

$$\delta_L = Max\left[\delta_{LN}^{c1},\ \delta_{LD}^{c2},\ \delta_{RD}^{c2}\right]$$

We use Equation 9 to calculate the saved energy (checkpoint time reduction), for Figure 18. Furthermore, δ_L corresponds to the `PHX` checkpoint time reported in the Figure 19 of §4.5.

4.5 Evaluation

Experimental Setup. Table 7 summarizes the characteristics of the experimental testbed used in the evaluations of Phoenix. We emulate PMEM characteristics on our experimental platform and run real-world HPC applications performing C/R. The file backed *mmap()* system call allocates specific memory regions that get treated as PMEM during `PHX` operations. We host the memory mapped file in *tmpfs* (in-memory file system). The mapped memory regions allow us to treat the emulated PMEM as virtual memory while the *tmpfs*-hosted backed file guarantees the data persistent over application restarts. We emulate the bandwidth characteristics of PMEM using software delays during the memory copy operations similar to [49]. The read/write data size (MB) divided by PMEM read/write bandwidth constants (MB/s) gives us the delay in seconds. We verify and calibrate our software delay mechanism by running the Stream benchmark [59] over our emulation code.

Table 7: Experimental setup details. We use Stampede compute nodes [53] for our experimental testbed

Number of nodes	4
Interconnect	InfiniBand Mellanox,56 Gbit/s
CPU	Intel Xeon E5 2.7 GHz
CPU cores per node	2 sockets, 8 cores/socket
Total Main memory	32 GB
Emulated PMEM	12GB
Effective Main memory	20GB

Table 8: Variable size distribution percentages (%).

Application	500K-1MB	10-20MB	50-100MB	above 100MB
CM1	40	0	54	4
GTC	45	9	0	45
S3D	50	25	0	25

Applications. We evaluate Phoenix with three real-world HPC applications and observe the C/R performance. The applications consist of algorithms related to different scientific fields, and each application shows different C/R characteristics such as compute to checkpoint data ratios, variable access patterns, etc. The working-set size is selected based on our computing resource capabilities.Figure 8 shows the checkpoint variable size distribution of each of those applications.

1. Gyrokinetic toroidal code (GTC) is a three-dimensional particle-in-cell application [35] used in micro-turbulence fusion device studies. The checkpoint data constitutes of 2D/3D arrays. We vary the *mpsi* value to obtain different checkpoint sizes.

2. CM1 [36] is a three-dimensional non-hydrostatic atmospheric model, used in studies of atmospheric phenomena. CM1 application saves a large number of variables ($\sim$ 50) compared to other applications during its checkpoints.

3. S3D is a direct numerical simulation (DNS) solver [60] that simulates the microscales of turbulent combustion. It solves the full compressible Navier-Stokes equations.

Experiment Overview. We run each of our HPC benchmarks against three different checkpoint schemes, (i) pmem - indicates a naive method of copying all data at once during checkpoints (ii) phx - indicates the proposed, bandwidth aware C/R method in this paper and finally (iii) phx-ec - indicates 'phx', with early-copy optimization. We record (i) the checkpoint times of each benchmark application with varying per-core-bandwidth values, (ii) the time spent on de-staging, and (iii) the compute iteration time. The results are presented in terms of per-core bandwidth. The metrics represent the effective bandwidth seen by one processor core during parallel PMEM writes and thus we consider the parallel

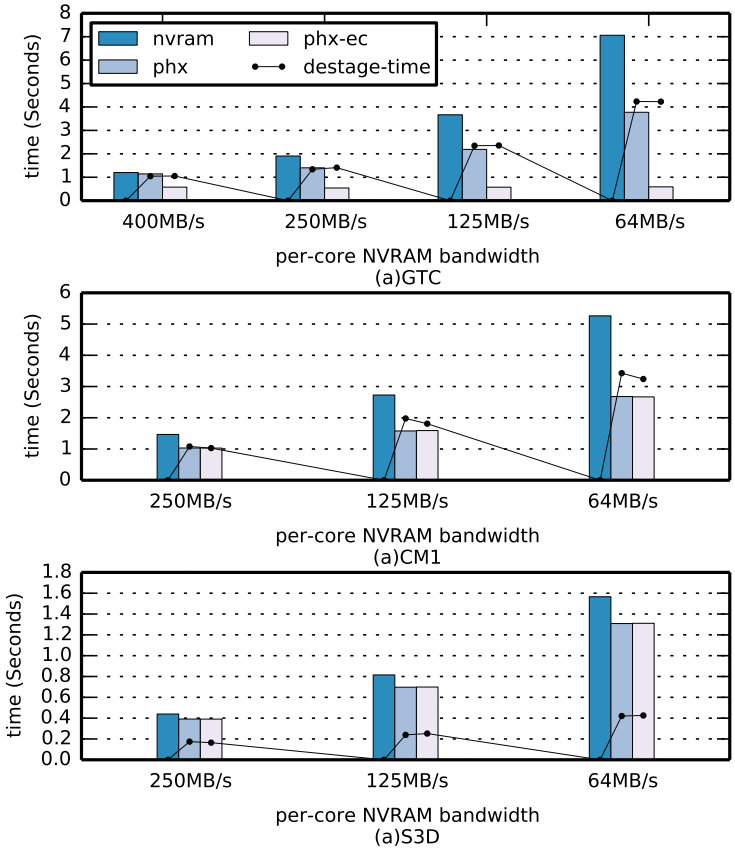

Figure 19: Checkpoint times of workloads, plot against varying per-core PMEM bandwidths (per-core NVRAM bandwdith in figure) and checkpoint schemes

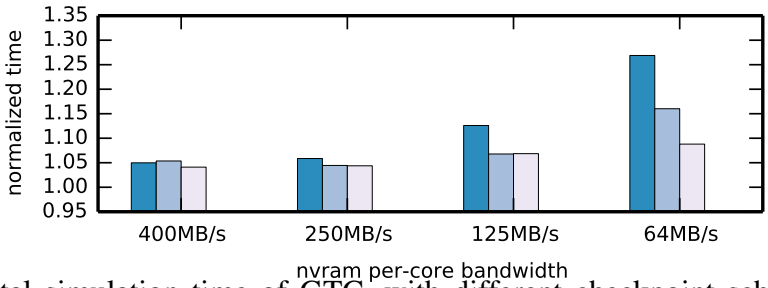

Figure 20: Total simulation time of GTC, with different checkpoint schemes, over ten iterations. Normalized to checkpoint free execution time of the same. This figure shares the legend-key with Figure 19

writes on the PMEM device. In all experiments we use the N=2 replication scheme for PHX checkpoints, and the buddy node is chosen to be one network switch away.

We run 12 MPI processes on each node. Each MPI process has a helper thread associated with it. However, all helper threads within a node are pinned to a single processor core. Thus, one core is dedicated for all the helper threads. We re-run the experiments with increasing number of helper cores in an attempt to see if the pinned core bottlenecks the performance, but it did not make any difference.

GTC Observations. GTC is a highly I/O (checkpoint) intensive application and therefore a naive PMEM data copy time can be as high as 37% compared to computation iteration of the application. In our experimental scale, the GTC benchmark workload writes 235 MB of checkpoint data per core, spends ~18 seconds in each of its compute iterations before executing a checkpoint at the cost of ~7 seconds for a per-core PMEM bandwidth of 64 MB/s (see Figure 19). Next, we run the same experiment with PHX C/R, while giving each core a staging buffer size of 125MB (50% of checkpoint data). PHX cuts the checkpoint time by half and measurements suggest that limited bandwidth of PMEM device still dominates the data movement overhead. The GTC variable access patterns, suggests that there is an opportunity for early copying of data. The resulting checkpointing scheme PHX-ec improves the checkpoint time by as much as ~12x over naive PMEM checkpoints.

CM1 Observations. The CM1 benchmark represents a highly compute intensive class of applications within our selection of benchmarks. Therefore, PMEM checkpoint time accounts for ~7%, compared to the compute iteration. Similar to the previous experiment, we start the experiment by allocating staging buffer budget of ~50% of the checkpoint data size. The PHX checkpoint scheme delivers up to ~2x (see Figure 19) faster checkpoint times over naive PMEM checkpoints. However, the PHX-ec scheme does not improve performance further. We attribute this behavior to CM1's data access patterns, where almost all of the checkpoint data are in the critical path of checkpointing. Thus, in the absence of early copy of variables, the PHX checkpoint speed up is loosely proportional to the allocated

staging buffer envelope.

S3D Observations. S3D checkpoints relatively smaller number of variables, 'yspecies' being the largest of them all. Application traces reveal minimal opportunity in terms of early copying of the variables – all the checkpoint data contributes to critical path checkpoint times. Our selected workload configuration produced ∼50MB of checkpoint data per core. Assigning a staging buffer size of 10MB (20% of checkpoint data) was enough for PHX to treat, all the variables except 'yspecies' as Ddata. PHX performs ~1.2x better (see Figure 19) than the naive PMEM checkpoint under this configuration. As PHX currently use variable granularity during Ddata and Pdata splits, the selected staging buffer size gives the most C/R performance for selected S3D checkpoint size.

Total Simulation Time. Next, we compare the total simulation time of GTC over ten iterations (see Figure 20), with the same checkpoint schemes. We show that PHX reduces the checkpoint time, thereby also reducing the overall simulation time. The execution times are normalized to a checkpoint-free run of the same application. Results show that PHX brings down the total checkpoint overhead up to ∼18% under 64MB per core PMEM bandwidth. It is important to note that the improvement margins are impressive, as the compute time dominates over the checkpoint times at the granularity of total simulation time.

Interference to Compute Iterations. The main compute iterations of the HPC applications constantly access the volatile memory for both instruction fetch and load/store of data. In addition, Phoenix moves large volumes of checkpoint data in and out of volatile memory for the (i) local staging of checkpoint data, for (ii) copying locally staged data to buddy DRAM, and for (iii) moving staged data into the PMEM (de-staging). Therefore, it is important to gauge the effect of the volatile memory usage in PHX on the main computation iterations of the HPC application.

We measure the compute iteration time between the checkpoints for each of our checkpoint schemes (not shown for brevity) and compare it with the failure-free, checkpoint-free

compute iteration time. The compute iteration times remain relatively constant across different checkpoint schemes for the failure-free and checkpoint-free execution.

4.6 Chapter summary

In this chapter we explore the bulk data movement challenges in HPC checkpoint I/O. Next generation exascale HPC applications are expected to generate an order of magnitude more data as application checkpoints. Solutions such as `NVStream` offer PMEM-aware memory I/O runtimes that make it possible for checkpoint I/O to be handled efficiently, However such solutions are still vulnerable to limited device bandwidth (a technology parameter) of PMEM. We present Phoenix (`PHX`) – a PMEM-bandwidth aware library for checkpoint I/O that deals with the limited PMEM bandwidth through simultaneous use of PMEM and local/ peer nodes' DRAM devices, thus increasing the effective data movement bandwidth. `PHX`'s PMEM-bandwidth-aware design lead to reduction in the time length of I/O operations in the critical path, associated with the slow PMEM device. Our `PHX` solution guarantees adequate reliability and persistence for DRAM resident object state by replicating them across peer nodes' memory via high-bandwidth interconnects. Experimental analysis using real-world HPC applications on emulated PMEM hardware shows that `PHX`'s controlled use of node-local and remote-node memory bandwidth, delivers up to $\sim 2\times$ and $\sim 12\times$ speed-up for checkpoint I/O for the CM1 and GTC HPC applications

CHAPTER V

CONCURRENT APPLICATION PROGRAMMING ON PERSISTENT MEMORY

The last two chapters focus on adopting PMEM into the existing persistent I/O abstractions with minimal application changes. There, we serialize/deserialize the applications' runtime memory structures into a durable format in the form of object/files/etc. However, with PMEM being both memory (byte-addressable) and persistent, we now can unify both runtime and durable state of our applications. Durable application programming on PMEM main memory is not easy. It further burdens the programmers with persistent data consistency semantics, in addition to existing challenges, including proper concurrency control in application design. We identify `transactions` as a suitable programming abstraction for handling both concurrency control and failure-atomic PMEM programming. In this chapter, we propose and evaluate different concurrency control and crash-consistency mechanisms that conform to the transactional notion and evaluate them against different PMEM configurations to draw valuable insights on the correct mix and match between them.

5.1 Introduction

One of the key appeals of persistent memory is that they allow applications to access storage directly using processor load and store instructions rather than relying on software intermediaries like file systems or a DBMSs [61]. However, ensuring that data stored in PMEM is always in a safe and recoverable state (i.e., the data is *crash-consistent*) is both hard and incurs performance overheads [62, 11, 13, 61]. To ensure crash consistency, application developers have to carefully orchestrate the movement of data from the volatile to non-volatile components in the memory hierarchy subject to recoverability constraints.

To exacerbate the challenge of ensuring crash consistency, different systems provide different guarantees on when data may be considered persistent. For example, Intel and Micron guarantee that data becomes persistent only when it reaches the memory controller of the PMEM device, i.e., the system's persistent domain includes the memory controller and the PMEM devices [63]. We refer to such systems as having *transient caches*. However, HPE's PMEM [64] guarantees that the entire cache hierarchy is persistent, i.e., the system's persistent domain includes the entire memory hierarchy. We refer to such systems as having *persistent caches*. Based on the systems' persistent domain, developers have to tailor their applications to achieve crash consistency. The diversity of PMEM applications [38] further complicates achieving crash consistency. Multi-threaded applications require developers to ensure correct synchronization on top of crash consistency.

One way to simplify the development of PMEM applications is to use a single abstraction that guarantees both crash consistency and correct synchronization. The transactional programming interface is particularly well suited for this approach. We have been using transactions to independently achieve crash consistency (e.g., database transactions) or proper synchronization (e.g., transactional memory systems). To this end, prior work has proposed numerous systems to simultaneously guarantee both these properties [11, 13, 14, 15, 16, 17, 18]. While these systems provide transactions with the desirable ACID properties that permit their use for both `crash-consistency` and `synchronization`, they do so in a myriad of ways; some implemented completely in software while some rely on hardware support, some use undo logging, while others use redo logging. So, developers are faced with a bewildering array of choices, with varied performance characteristics that change with applications and the system used. For a platform with given hardware features and consistency and synchronization requirements, is it possible to streamline the design space and quickly arrive at a correct and performant implementation of a transactional system?

To efficiently describe the properties of a system providing both `crash-consistency`

and correct synchronization simultaneously for the same program data, we coin the term crash-sync-safety (§5.3). We use crash-sync-safety to characterize transaction systems with respect to their crash-consistency requirements, impacted by the persistence domain, and synchronization requirements, determined by the support for and use of concurrency. We perform a detailed study of systems with different characteristics under various operating scenarios and provide an understanding of the relationship of crash-sync-safety and the best implementation of a transactional programming model for a given PMEM system.

We implement and evaluate various methods for realizing the transactional programming model, by following closely the best designs presented in prior work: hardware transactional memory (HTM) [17], software transactional memory (STM) [11], and undo/redo logging with locks [21, 11]. Through emulation and simulation, we consider both transient and persistent caches and vary the latency characteristics of the PMEM. Finally, we perform our characterization study on real hardware using the recently released Intel's DC Optane Persistent Memory [65].

Overall, in this chapter;

- We introduce the notion of crash-sync-safety and discuss how different system features (e.g., persistence domain, HTM support) and requirements (e.g., multi- vs. single-threaded use) impacts the implementation of this property.

- We compare the performance of various systems guaranteeing crash-sync-safety on systems with different persistent domains and different concurrency support and requirements.

- We show that there is no one best way to ensure adequate transactional support in PMEM-based systems and in fact, the best way changes with the crash-sync-safety property of the system, impacted by its persistence domain, the presence and efficiency of hardware support for transactional concurrency control, and the application

requirements.

5.2 Crash-consistency and synchronization

In order to illustrate the complexity of the design space, we briefly survey different implementations for crash-consistency (§5.2.1), for transaction synchronization (§5.2.3), and the impact of the hardware persistence domain on the relationship between the two (§5.2.2).

5.2.1 Crash-consistent transactions

Crash-consistent (failure-atomic) transactions ensure that a group of updates to PMEM locations performed by an application persist atomically, i.e., either all of them are observable or none of them are observable after a failure. Transactions are specified using `tx_begin()` and `tx_end()` calls. All the updates to PMEM between those two successive calls are guaranteed to persist atomically. For example, in Table 9, the updates to pA and pB are crash-consistent. *Crash-consistency* is generally achieved using undo or redo logging.

undo logging is a crash consistency technique that provides failure atomicity by undo-ing (or rolling back) changes from an aborted failure-atomic transaction. To be able to roll back changes, `undo` logging systems create an `undo` log entry *prior* to every update performed within the transaction. The `undo` log entry contains the current value of the memory location/variable that is being updated. Once the log entry had been created and persisted, only then is the actual memory location/variable updated. If a transaction succeeds, all the memory locations modified within the transaction are persisted and then a commit message is atomically persisted to the log to invalidate the log entries belonging to the transaction. If a transaction fails, during the recovery process, all valid log entries are used to roll back partial changes from a transaction. As shown in Table 10, `undo` logging systems must ensure that: (1) within a transaction, log entries must be created and persisted prior to every update and (2) at the end of the transaction, all memory locations modified within the transaction must be persisted before transaction commit.

redo logging is a crash-consistency technique that provides failure atomicity by redo-ing (or rolling forward) changes from committed failure-atomic transactions. To be able to roll forward changes, `redo` logging systems create a `redo` log entry for every update within the transaction. The `redo` log entries contain the latest updates while the actual data is maintained at a prior crash-consistent state. All the read requests for the memory locations updated within the transaction are serviced from the `redo` log. If a transaction succeeds, a commit log entry is created and persisted in the `redo` log, marking the commit of the transaction. In the event of a failure, the `redo` log entries of committed transactions are used to roll forward the application's data to its most recent crash consistent state. The log entries of uncommitted transactions are simply discarded. Periodically, the redo log can be truncated to reduce read indirections and to reduce the number of redo log entries that have to be applied during recovery. As shown in Table 10, `redo` logging systems must ensure that: (1) within a transaction, a `redo` log entry must be created for every update within the transaction and read requests to these locations must be re-directed to the log, and (2) at the end of the transaction, all the redo log entries and a commit log entry must be persisted.

5.2.2 Persistent and transient caches

There is much diversity among the types of PMEM technologies that are available on the market, as different vendors provide different performance characteristics and persistence guarantees. For example, HPE offers a battery-backed DRAM solution [66]. As this design is based on DRAM, the exposed PMEM's latency and bandwidth are as good as for DRAM. In addition, the battery can extend the persistency domain to the entire memory hierarchy, including CPU caches. **Persistent caches** ensure that all modified cache lines are effectively persistent. On the other hand, Intel and Micron's proposed 3D XPoint technology [22] has higher latency and lower bandwidth than DRAM, while the persistent domain includes only the memory controller, but not the CPU caches [63]. In case of a power failure, **transient caches** will lose modified data not already written back to the memory

Table 9: Threads executing dependent transactions. Correct implementations ensure that pA persists before pD, crash consistency might be violated otherwise.

THREAD-1	THREAD-2
tx_begin();	
pA = x;	tx_begin();
pB = y;	**if (pA == x)**
tx_end();	**pD = z;**
	pC = w;
	tx_end();

controller. So, the transaction system developer must use the appropriate instruction sequences to ensure that data becomes persistent on different hardware platforms.

5.2.3 Transactional memory

Transactional memory [67, 68] is used to synchronize the access of multiple threads to shared program data. Programmers enclose the critical code blocks with `tx_begin()` and `tx_end()` calls. Transactional memory guarantees the atomic execution of a transaction, using speculation. If the runtime detects a conflict with another transaction, it aborts one of the transactions, discards its speculative state and rolls back its execution to the `tx_begin()` call. Software transactional memory (STM) [68] is implemented using fine grained locking and write set logging in software. Hardware transactional memory (HTM) [67] is implemented using the L1 cache to buffer speculative writes and the cache-coherency protocol to detect conflicts with other threads. Current HTM implementations, such as Intel Transactional Synchronization Extensions (TSX), are best effort – transactions could abort for any reason, such as exceeding the L1 cache capacity, using unsupported instructions, or due to interrupts. Therefore, HTMs require a fallback mechanism to ensure progress, usually implemented using locking.

Table 10: Undo vs Redo logging; Undo logging suffers from frequent cacheline flushes and sfences while redo logging suffers from read-indirection overheads.

	Baseline Tx	**Undo Tx**	**Redo Tx**
1	tx_begin();	tx_begin();	tx_begin();
2	pA = x;	log[&pA] = pA; clwb(log[&pA]); sfence; pA = x;	log[pA] = x;
3	y = pA;	y = pA;	y = (log[pA] || pA);
4	pB = z;	log[&pB] = pB; clwb(log[&pB]); sfence; pB = z;	log[pB] = z;
5	tx_end();	persist_write-set(); commit_log();	clwb(log[pA]); clwb(log[pB]); sfence; commit_log();
6		tx_end();	replay_log(); persist_write-set();
7			tx_end();

5.3 Crash-sync-safety

In this work, we focus on applications that use a transactional programming model to get ACID guarantees. For example, in Table 9, updates within each transaction need to provide all or nothing semantics when the data gets to PMEM. Providing ACID guarantees requires that the transactional system correctly implement three components: (1) *crash consistency*, (2) *synchronization*, and (3) *composability*. `undo` and `redo` logging can be used to implement crash-consistent transactions for single-thread applications, but do not ensure the correct synchronization of multi-threaded applications, as shown in Table 11. Conversely, transactional memory or locking can be used to implement transactions that provide correct synchronization for multi-threaded applications, but cannot ensure crash-consistency for these transactions in case of a failure, as shown in Table 11. Simply guaranteeing these two properties is not sufficient to provide ACID guarantees. A transactional system guaranteeing to also ensure that these techniques compose by tracking dependencies between updates made by different threads *and* to persist the updates in the correct (execution) order. A correct implementation of the above three properties ensures ACID guarantees and we call such implementations `crash-sync-safe` (Table 11).

Table 11: Crash-sync-safety combines both crash-consistency and synchronization.

	Failures	Multi-threading
Crash-consistency	✓	✗
Synchronization	✗	✓
Crash-sync-safety	✓	✓

Programmers identify regions of code within their applications as transactions using `tx_begin()` and `tx_end()`, and are assured of the failure-atomicity of the updates within any transaction. Furthermore, all the updates within the transaction become atomically visible to any other thread in the system once persisted, and conflicting transactions (transactions accessing common memory locations with at least one of the accesses being a write)

execute in isolation. So, each transaction behaves as both a traditional transactional memory transaction and also a failure-atomic transaction. `crash-sync-safety` guarantees the following properties:

- **Property-1:** All the updates within a transaction are guaranteed to become visible atomically to other threads.
- **Property-2:** All the updates to PMEM locations within a transactions are guaranteed to be failure-atomic.
- **Property-3:** Conflicting transactions execute in isolation and updates to PMEM within conflicting transactions persist in respective transaction commit order.

5.4 Achieving crash-sync-safety

This section describes different implementations of a transactional library that ensures `crash-sync-safety` in detail. First, to achieve proper synchronization, transactions may be implemented using one of three broad approaches: (1) Hardware Transactional Memory (HTM), (2) Software Transactional Memory (STM), or (3) global locking. Each of these approaches can further be extended to additionally provide crash-sync-safety for transactions. Depending on whether the system has transient or persistent caches, the implementation details will vary. Next, we describe these different implementations (Table 12).

5.4.1 Crash-sync-safe HTM

Hardware Transactional Memory (HTM) offers atomicity and isolated transactions for volatile memory. With persistent memory systems, HTM implementations can be extended to ensure that they become `crash-sync-safe`. Designing crash consistent HTM (ccHTM) requires augmenting HTM with a separate `undo/redo` log in persistent memory [14] and logging data modifications within a transaction. It is important to note that the software fallback path must also be made crash consistent through appropriate logging. In the event of a failure, the ccHTM logs can be used to restore the application's persistent data to the

Table 12: Crash-consistency and crash-sync-safety implementations for single- and multi-threaded applications. ST: Single-threaded, MT: Multi-threaded, CC: Crash-consistent, Sync: Synchronization, CSS: Crash-sync-safety, TC: Transient caches and PT: Persistent caches. Techniques evaluated for single-threaded applications need to provide only crash consistency. Techniques evaluated for multi-threaded applications provide synchronization too, by using a spinlock where necessary. We note that the HTM+undo/redo implementations for transient caches are only approximating a crash-sync-safe solution.

	ST – CC		MT – Sync.	MT – CSS	
	TC	PC		TC	PC
seq	✗	✗	✗	✗	✗
HTM+seq (+spinlock)	✗	✗	✓	✗	✗
undo/redo (+spinlock)	✓	✓	✓	✓	✓
HTM+undo/redo (+spinlock)	approx.	✓	✓	approx.	✓
ccHTM+undo/redo (+spinlock)	✓	N/A	✓	✓	N/A
STM	✗	✓	✓	✗	✓
ccSTM	✓	N/A	✓	✓	N/A

most recent consistent state. While many different ccHTM implementations have been proposed recently [14, 15, 16, 17, 18], to the first order, they are all similar. In this work, we developed and implemented our own ccHTM design (see Figure 21), as a representation of the prior proposals.

Transient caches. For transient caches our `ccHTM` implementation augments a regular HTM [69] with support for failure-atomicity when modifying data in PMEM. Apart from the usual read/write set tracking employed in HTM designs, ccHTMs employ a separate write-set log in PMEM to maintain failure-atomicity. To update the PMEM log, we extend the hardware to issue write-combined, non-temporal stores (those that bypass the cache hierarchy, like x86's `movnt` [70]) for every write within a transaction. These writes are non-transactional operations [71], so they do not become part of the transaction's write set. Note that reads and writes that are part of the transaction use the regular temporal load/store instructions and are served from the CPU caches. We use `redo` logging in our `ccHTM` implementation, but `undo` logging would be similar. When used inside a hardware transaction, the `redo` log does not suffer from read indirection, because the values can be found as

speculative values in the L1 cache. At commit time, we first ensure that the log writes are persistent, then atomically persist a log commit message in the PMEM log, then make the transaction's updates visible to other cores in the system. Overall, transactions first attempt an execution as a failure-atomic hardware transaction. However, if a hardware transaction aborts, the fallback path involves acquiring a global lock and executing pessimistically using a software undo/redo log. Next, we describe in detail the various aspects of our `ccHTM` implementation.

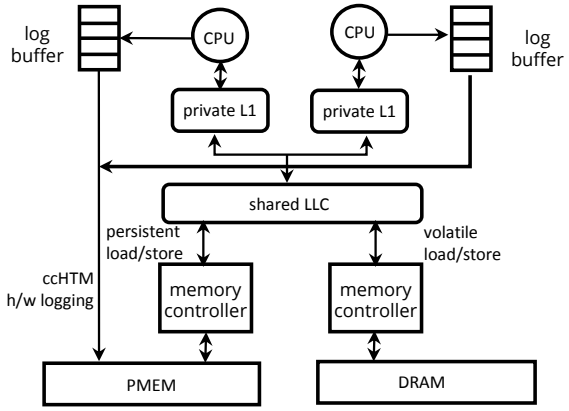

Figure 21: Design of `ccHTM`

Persistent write-set logging. HTM runtimes keep track of the read/write sets of the executing transaction. Writes executing inside a transaction are held in the L1 cache in speculative state, isolated from the rest of the memory hierarchy until the successful completion and commit of the transaction. Similar to HTMs, `ccHTM` issues writes in the transaction to the L1 cache. In addition, `ccHTM` intercepts each write and augments it with a hardware based non-temporal log-write request into a thread-local PMEM log. So, every write within a transaction results in a temporal write to the L1 cache and a non-temporal log write. The PMEM log write is asynchronous to the intercepted transactional write, thus has minimal impact on the transaction's critical path. The `ccHTM` log is durable and atomic updates do not suffer from inherent read indirection overheads of logging, as incoming read requests are being served directly from the L1 cache.

Transaction commit. An ccHTM transaction comprises both volatile state (the cache lines held speculatively in the L1 cache) and persistent state (the ccHTM-log). Thus, the ccHTM transaction commit sequence differs from a traditional HTM commit. The ccHTM commit sequence includes: (1) updating the book-keeping structures of speculative cache-lines, (2) failure-atomic write-set log commit on PMEM, and (3) atomically releasing the speculative cache-lines to the rest of the memory hierarchy. It is important to note that ccHTM commit operation combines two commit phases – a persistent memory commit, in the form of ccHTM-log commit, and a volatile memory commit, in the form of speculative cache-line unlocking. A ccHTM log commit involves two sfence instructions. The initial sfence drains the buffered asynchronous log writes to the ccHTM-log. Next we atomically persist/update the ccHTM-log's tail-index with the latest log-entry index value, followed by another sfence. The ccHTM-log's tail-index update doubles as a commit-flag entry and enables fast log truncation. Once the transaction has been committed in PMEM, the volatile commit is performed by atomically moving the affected cache-lines out of the speculative state. Once the volatile commit is performed, the transaction has successfully completed.

Transaction abort. Similar to HTM aborts, ccHTM aborts may be triggered due to (but not limited to) a load/store on another thread that conflicts with the current transactions' write/read set, OS interactions like system calls or context switches, L1 cache capacity overflow. In addition to all of the HTM abort causes, ccHTM transactions abort if the runtime runs out of ccHTM-log space during hardware logging. We introduce a new abort flag called NO_LOG_SPACE to capture this abort cause. A transaction abort includes (1) discarding speculative L1 cache-lines, and (2) invalidating ccHTM-log appends. We rely on existing HTM capabilities to achieve (1). We do not explicitly invalidate ccHTM-log entries as they remain invalid till ccHTM-log's tail-index update.

Fallback path. Similarly to hardware transactions, ccHTM transactions are best effort – the transactions are not guaranteed to complete. ccHTM transactions use regular write-ahead-logging (WAL) on the fallback path to ensure persistence. The fallback path transactions

use either `undo` or `redo` logging while HTM transactions use `redo` logging. In addition, a global lock ensures the synchronization between the fallback path software transactions and `ccHTM` hardware transactions. To ensure isolation, the hardware transactions read the global lock as soon as they start executing, which makes them abort if another thread acquires the lock.

Log truncation. We truncate the transaction logs (both ccHTM and fallback path) at the end of each transaction – eager log truncation. With `redo` logging, log truncation involves first persisting the cache-lines modified as part of the transaction and then invalidating the transaction's log entries. We truncate the ccHTM transaction logs as part of the transaction commit step, i.e., once the PMEM log of the transaction is committed, we perform the following steps: (1) issue `clwb` requests to all the cache-lines in the write-set, (2) issue an `sfence` to ensure their writeback, (3) issue a non-temporal update request to atomically reset the `ccHTM`-log's tail-index to truncate/invalidate all the previously written log entries, and (4) issue another `sfence` to ensure that the update request has been persisted. Once these four steps are performed, the volatile commit of the transaction is carried out. We also truncate the logs for the transactions executed in the fallback (software) path as soon as they commit. It is important to note that since both the fast path and slow fallback path employ log truncation, it is feasible to employ different logging techniques in the different paths. For example, it is possible to use `redo` logging in the fast path and `undo` logging on the fallback path. This design approach allows us to evaluate `crash-consistency` mechanisms that use different logging techniques on the different paths. Furthermore, this eager log truncation approach, relives our `ccHTM` implementation of the burden of tracking the execution order of different transactions in their respective PMEM logs, as is necessary in other prior approaches [16].

Persistent caches. In systems with persistent caches, speculatively updated ccHTM cache-lines are persistent as soon as they are atomically released. (when they made visible in L1 cache). However, transactions executing in fallback path still need atomic updates in the

form of undo/redo logging. So, regular HTM implementations can be augmented with a fallback path log and can ensure crash-sync-safety with no additional changes to the HTM.

5.4.2 Crash-sync-safe STM

Software Transactional Memory (STM) offers atomicity and isolated transactions for volatile memory. All the data modifications made within the transaction are made visible to other threads atomically when the transaction commits. If the transaction aborts, none of the data modifications made within the transaction become visible. STM implementations track the read and write sets of individual transactions to ensure transaction atomicity. Further more, they provide transaction isolation by detecting conflicting transactions that modify at least one common memory location and aborting some of them as necessary.

With persistent memory systems, STM implementations can be extended to ensure that they become crash-sync-safe, i.e., data modifications within a transaction will persist atomically and the dependencies are handled properly (§5.3). There are two broad approaches to designing crash consistent STM (ccSTM): (1) augment STM with a separate `undo/redo` log in persistent memory [11, 13] or (2) repurpose the write sets already maintained as part of the STM implementation to also function as a `undo/redo` log. In the event of a failure, the ccSTM logs can be used to restore the application's persistent data to the most recent consistent state. In this work, we concentrate on ccSTM designs that maintain a separate `undo/redo` log, similarly to [11].

Transient caches. In systems with transient caches, in order to make sure their log entries are persistent (`undo` or `redo`), ccSTM designs have to write back the log entries from the processor caches to the memory controller. Furthermore, log entries have to be written back as per the ordering constraints of the logging mechanism employed (as discussed in § 5.2.1) using carefully orchestrated `clwb`, `sfence`, and non-temporal store instructions (e.g., `movnt`).

Persistent caches. However, in systems with persistent caches, ccSTM log entries are

persistent as soon as they are created (when they reach the L1 cache). So, regular STM implementations also ensure crash consistency with no additional changes in systems with persistent caches.

5.4.3 Crash-sync-safe locking

This implementation of transactions acquires a global spinlock at the beginning of every transaction and releases it at the end of every transaction. While this naive implementation suffers from frequent false conflicts for multi-threaded applications, it does offer one advantage. It is a very light-weight approach when no concurrent transactions are executed by an application, an extreme case of which is a single-threaded application. Mostly, we use this design point for the sake of completeness in our `crash-sync-safety` design space analysis. While global locking achieves proper synchronization, to achieve `crash-sync-safety`, transactions are usually extended with either `undo` or `redo` logging, which we describe next.

Transient caches. In systems with transient caches, data can be considered persisted only once it has been written back from the volatile cache hierarchy to the memory controller using one of `clflush`, `clflushopt`, `clwb` instructions. Further more, some of these instructions are non-blocking, so applications need to issue a subsequent `sfence` to ensure that the instructions have been fully executed and the associated data is actually persistent.

`undo` logging systems have to ensure that log entries are persistent before they can allow actual memory locations to be modified within a transaction. As shown in Table 10, `undo` logging systems use a combination of `clwb` and `sfence` instructions prior to every data update, i.e., every store instruction. This frequent use of blocking `sfence` instructions could result in severe performance degradation. `redo` logging systems have to ensure that all the redo log entries and the commit log entry are persisted by the end of a transaction. Further more, the commit log entry may persist only after all the redo log entries have been persisted. As shown in Table 10, `redo` logging systems use a combination of `clwb` and

`sfence` instructions within a transaction.

Persistent caches. However, in systems with persistent caches, data is considered persistent as soon it has been written to the L1 data cache. So, no writeback of data to the memory controller is necessary on such machines. On systems with persistent caches, `undo` logging implementations need to ensure that log entries are created before the data update, while `redo` logging implementations need to ensure that redo log entries are created for every update within the transaction and that the commit log entry is created before the completion of the transaction. Since x86 systems guarantee TSO, the program order of stores ensures that the stores belonging to the log entry creation and data update are performed in order, without the need for any intervening `sfence` or `clwb` instructions. For example, with persistent caches on an x86 machine, all the `clwb` and `sfence` instructions shown in Table 10 become obsolete. However, for systems with a weaker memory model (e.g., ARM), an appropriate `fence` instruction is necessary to ensure that the stores are executed in program order. Persistent caches significantly improve the performance of `undo`/`redo` logging systems as they eliminate expensive `clwb` and `sfence` instructions.

5.5 Implementation and evaluation methodology

We want to understand the overheads of crash-consistency and crash-sync-safety, as well as what is the best implementation of a transactional library that provides these properties, given various PMEM characteristics, persistence domains, and workload characteristics. To do so, we compare the performance of different transactional library implementations along the following axes: (1) Persistence domains of the system. Specifically, we consider systems with *persistent* and *transient* caches (§5.2.2). (2) Single-threaded vs multi-threaded applications. Single-threaded applications just require crash consistency while multi-threaded applications require crash-sync-safety (§5.3). (3) Evaluation platform – real hardware with Intel Optane DC PMEM or an architectural simulator.

Real Hardware vs Architectural Simulation. To evaluate `ccHTM`, we use two different

platforms: (1) bare-metal hardware with TSX and Intel Optane PMEM and (2) SESC, a cycle-accurate simulator. While neither approach allows us to accurately evaluate `ccHTM`, they complement each other and provide a comprehensive analysis of the competing mechanisms. For example, simulation accurately models the proposed hardware changes not possible with TSX. However, real hardware more accurately factors in transaction abort rates introduced due to system jitter (background activities, thread context switches, cache capacity constraints) and the latency and bandwidth constraints of real PMEM DIMMs.

Intel Optane PMEM hardware testbed: We use a Intel Xeon server (Cascade Lake microarchitecture) with 96 cores over 2 NUMA sockets. We use Fedora Linux as the OS. The Intel processor supports restricted transactional memory (`rtm`) and the cache-line-write-back instruction (`clwb`). Each processor socket has access to 375 GB of DRAM and 756 GB of Intel Optane PMEM, configured in Direct Access Mode [1]. The PMEM memory is managed by a DAX supporting file-system, hence applications have to explicitly map PMEM memory into their process address space prior to using the PMEM. Therefore, we modify our applications to explicitly allocate all memory dynamically from the PMEM address space using the `libvmem` allocator from the Persistent Memory Development Kit (PMDK) [21]. We allocate persistent memory (mmap) from the closest PMEM DIMM (NUMA aware). We bind each of the application threads to a compute core. Thread binding prioritizes the compute cores within the same NUMA socket and assigns compute cores from a different socket only when an application uses up all the cores in the current socket. Out of five runs, we report the mean of the middle three runs.

Simulator: We implemented `ccHTM` as an extension to SESC-HTM [72], which emulates the instruction behavior(commit, abort, etc.) within the HTM_begin() and HTM_end() code regions and passes them to a back-end timing module for simulation. We augment the writes happening within a HTM transaction with an asynchronous, non-temporal log-write to the PMEM resident log, in addition to the temporal L1 cache write. Furthermore we implement the `clwb` and `sfence` instructions necessary for the correct functioning of

Table 13: Simulator config, adopted from [72]

Processor	16 cores 1 GHz connected via bus-interconnect
L1 cache Ins & Data	64 KB per core / 64 B cacheline/ 8-way set associative
L2 cache Ins/Data	256 KB per core/ 64 B cacheline/ 8-way set associative/ hit-latency 18 cycles
L3 cache	16 MB shared/ 64 B cacheline/ 16-way/ hit-latency 34
Coherence protocol	MESI across L2 caches
PMEM log size	10 MB per core/thread
PMEM r/w latency	250/750 cycles

software-based crash-consistency mechanisms. Table 13 lists the configuration of the various hardware structures modeled in our simulator. Since SESC was designed for MIPS, we cross compile the STAMP benchmarks and the `ccHTM` library into a MIPS binary.

Workloads. We use two benchmarks from the PMDK project [21], namely C-tree and Hashmap. These two benchmarks implement a persistent crit-bit tree and a hashmap. We port these two applications to use different persistent memory transactional mechanisms. We run the `pmembench` workload generator provided with PMDK and use workload parameters from [38]. In addition, we use the transactional applications from STAMP [73], a popular benchmark suite used by others to evaluate libraries for PMEM [38, 74, 75]. We augment transactions with crash-consistency, on top of the atomicity, consistency, and isolation guarantees already provided. To better understand these workloads, we instrumented the simulator to count the load/stores for each transaction.

5.6 Evaluating crash-sync-safety

In this section, we seek to understand the cost of implementing crash-sync-safety (§5.3) in various ways. To do so, we evaluate multi-threaded applications that provide both `crash-consistency` and `synchronization`. We use the crash-consistency mechanisms in Table 12. We use a spinlock to ensure correct synchronization for the `undo`/`redo` logs and on the fallback path of the `HTM`. We want to answer the following questions: (1) What

is the most efficient implementation of crash-sync-safe transactions? To answer this question, we compare HTM-based crash-sync-safe transactions with STM-based crash-sync-safe transactions and with undo/redo logging using a spinlock. (2) What is the overhead of achieving crash-sync-safety? To answer this question, we compare to a sequential implementation baseline, with no `crash-consistency` and no `synchronization`. (3) What is the overhead of crash-consistency for multi-threaded applications that are properly synchronized? To answer this question, we compare with a non-crash-consistent baseline (HTM+spinlock). (4) How does the persistence domain of the PMEM influence the results? To answer this question, we consider two different PMEM devices: one with transient caches (§5.6.1) and one with persistent caches (§5.6.2). Current HTM for systems with transient caches ensure proper `synchronization`, but not `crash-consistency`, so our real hardware evaluation on transient caches only approximates a crash-sync-safe implementation based on HTM. Therefore, we use simulation to properly evaluate the overheads of the crash-sync-safe HTM.

Summary of crash-sync-safety results. We evaluate multiple transactional implementations on real hardware with the new PMEM devices, as well as using an architectural simulator. Our results are summarized below. (1) We find that ccHTM consistently outperforms other transactional implementations, by 0.06X-30X (at 8 threads) for transient caches, and by 3X on average for persistent caches. The only exceptions are applications which are known for being problematic for hardware transactions, i.e., with large read or write sets that overflow the cache, or with unsupported instructions that always abort the hardware transactions. Therefore, extending HTM with `crash-consistency` for transient caches is the most promising solution to provide crash-sync-safe transactions. The simulation results show that making the HTM crash-consistent does not add significant overhead compared to a non-crash-consistent HTM (HTM+spinlock). For persistent caches, current HTMs (e.g., TSX) are already crash-sync-safe. (2) When using ccHTM to achieve crash-sync-safety,

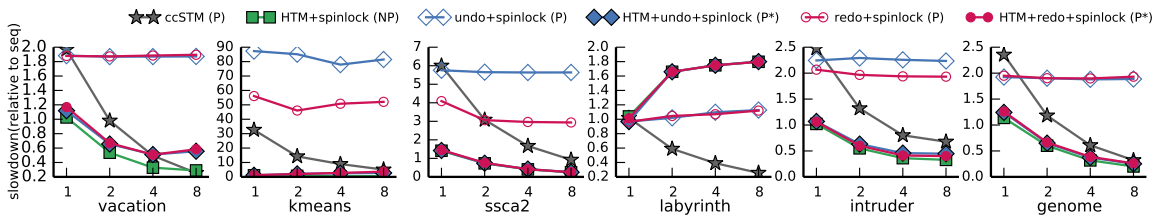

Figure 22: TSX-enabled hardware with real PMEM and transient caches by number of threads (X axis). (P) crash-consistent; (NP) not crash-consistent; (P*) approximates crash-consistent solution.

it comes almost for free. The overheads of crash-consistency are subsumed by synchronization overheads and, as applications scale, performance increases compared to single-thread execution. When the ccHTM implementation does not achieve scalability due to aborts, the ccSTM still ensures this property. (3) If we disregard scalability improvements given by running multiple threads, we can measure the cost of crash-consistency compared to a non-crash-consistent solution that still ensures the synchronization. On average, HTM+undo(redo) is 2.3X (2.4X) slower than HTM+spinlock (for 4 threads), for transient caches. When caches are persistent, this cost becomes negligible. (4) In multi-threaded applications, the persistence domain still plays a very important role, but the results are not as dependent on it as they are for the single-threaded applications. The overhead of crash-consistency is considerably lower when caches are persistent. In addition, HTM is `crash-consistency` out of the box, so no changes are necessary.

5.6.1 Transient CPU caches

We use our real test-bed and architectural simulator to evaluate the cost of crash-sync for transient caches.

Real. Figure 22 shows the scalability of the various approaches outlined above with varying number of threads, on real hardware, using Intel TSX for the `HTM`. `HTM+undo` (`HTM+redo`) outperforms `undo` (`redo`) for vacation, kmeans, ssca2, intruder and genome by 3.6× (3.7×), 30.8× (18.3×), 13.3× (7.1×), 0.6× (0.6×) and 4.9× (4.8×), respectively, at 4 threads. The only exception is labyrinth, where many transactions overflow

the cache and abort, ending up executing on the fallback path (undo/redo), serialized by a spinlock.

Compared to the ccSTM, HTM+undo (HTM+redo) is faster for vacation, kmeans, ssca2, intruder and genome by 1× (1×), 3.5× (3.2×), 3.9× (4.0×), 1.7× (1.9×) and 1.6× (1.5×) respectively and slower on labyrinth by 53.06% (53.10%) at 4 threads. Choosing between software and hardware transactions largely depends on the workload, especially the size of the transactions, conflict rate and usage of TSX-unsupported operations. ccSTM performs better on workloads with larger transactions (e.g., labyrinth) or more contention and scales better to a larger number of threads. We attribute this behavior to its better conflict resolution. However, for small transactions, the software conflict detection and resolution of the ccSTM introduces too much overhead, which is greatly reduced by the simpler hardware-based requester-wins policy of the HTM.

We approximate the cost of crash-consistency for multi-threaded applications by comparing to HTM+spinlock, which suffers from the overheads of synchronization, but not crash-consistency. HTM+undo and HTM+redo are, on average, 2.3× and 2.4× slower than HTM+spinlock for all workloads (at 4 threads).

As in the single-thread applications, the choice between undo and redo logs greatly depends on the workload characteristics. However, when we use HTM on the fast path, the differences between undo and redo logs on the fallback path are significantly diminished with HTM+undo and HTM+redo resulting in similar performance.

We compare the ccSTM with an STM to understand the cost of crash-consistency for the STM. ccSTM is at most 8.2× slower than an STM with no crash-consistency for 1 thread and at most 7× slower for 8 threads. We see that while crash-consistency definitely adds a noticeable overhead, it does not impact the scalability of the original STM. Moreover, the difference between ccSTM and STM decreases with increasing the number of threads, showing that the overhead of fences is amortized between multiple concurrent

threads. Finally, the cost of crash-consistency does not tell the entire story, as we provide both crash-consistency and synchronization for multi-threaded applications. Thus, we also measure the cost of crash-sync-safety by comparing with a baseline with no crash-consistency and no synchronization (seq). While crash-consistency adds overhead compared to volatile in-memory execution, efficient synchronization often improves performance by enabling the application to scale to multiple threads. Therefore, the cost of crash-sync-safety is much lower than the cost of crash-consistency when we can pair efficiently synchronization and crash-consistency, using STM or HTM. However, when we use distinct methods for the two, the overheads compose and the cost of crash-sync is much higher, as exemplified by undo/redo using spinlocks being $.9\times/1.4\times$ and $2.7\times/2.9\times$ expensive than ccHTM+undo/redo and ccSTM on-average.

As in §5.7.1, these numbers are an approximation of ccHTM results, as only the fallback path is crash-consistent. We evaluate a full-fledge ccHTM in the simulator in the next section.

Simulated. Figure 23 shows ccHTM results using simulation. We use this to confirm that adding crash-consistency on the HTM fast path does not hinder its performance. Once again, the general trends from real hardware largely hold in the simulated environment as well: (1) HTM+undo/HTM+redo comfortably outperform undo/redo. For example, for Vacation benchmark with 4 threads, the improvements are $3\times$ and $6\times$ respectively. The only exception to these general trends are seen in the case Labyrinth, where undo and redo perform better than HTM based approaches due to the high transaction abort rates inherent to the workload. (2) undo/redo exhibit the highest overheads and poor scalability due to the spinlock. (3) All crash-consistency mechanisms increase execution time over the non-crash consistent baseline, HTM+spinlock. However, on-average ccHTM+spinlock increases execution time by only 8% compared to HTM+spinlock. The simulation results differ from the real hardware results mainly in the scalability showed by ccHTM. This difference comes from the system events that occur in the real systems, but are hard to model in a simulation

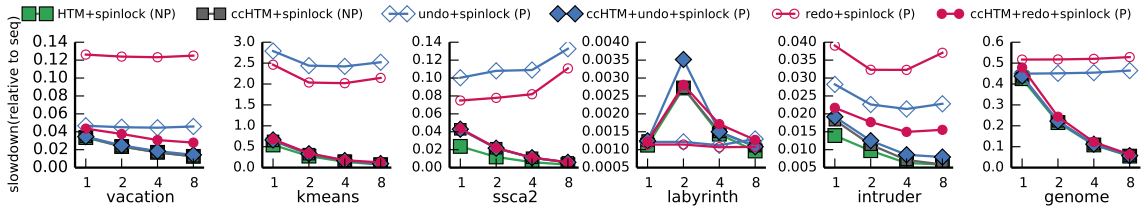

Figure 23: Simulation, transient caches by number of threads (X axis). (P) crash-consistent; (NP) not crash-consistent; (P*) approximates crash-consistent solution.

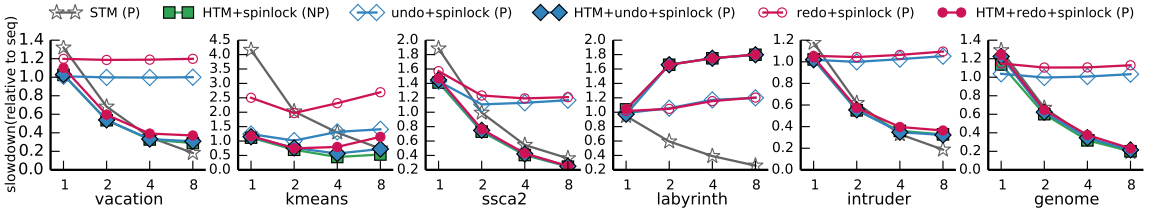

Figure 24: TSX-enabled hardware with real PMEM and emulated persistent caches by number of threads (X axis). (P) crash-consistent (NP) not crash-consistent.

environment.

5.6.2 Persistent CPU caches

We use our bare-metal test-bed to emulate persistent caches. TSX ensures `crash-sync-safe`. We show the results in Figure 24. As expected, `undo` and `redo` perform the worst and exhibit poor scalability because they serialize all transactions using a global spinlock. `HTM+undo` (`HTM+redo`) is 2.4× (2.4×) faster than `undo` (`redo`) on average for all workloads at 4 threads. The only exception is labyrinth, which causes frequent transaction aborts due to overflows.

While `STM` incurs high overheads in low contention scenarios (1 or 2 threads) or when transactions are small, it exhibits good scalability due to its fine-grained locking and generally performs the best at 8 threads and for large transactions. `STM` is 1.6× (2.0×), 6.9× (6.9×) faster than `HTM+undo` (`HTM+redo`) for vacation and labyrinth at 8 threads.

5.7 Evaluating crash-consistency

In this section, we seek to understand what is the cost of `crash-consistency` for single-thread applications, i.e., when applications do not require synchronization. To do so, we perform an exhaustive study using multiple hardware platforms, using simulation and emulation when the actual hardware is not available. We evaluate both transient (§5.7.1) and persistent caches (§5.7.2). We present the results relative to a sequential execution baseline (`seq`) that does not provide `crash-consistency` nor `synchronization`. We evaluate the crash-consistency mechanisms described in Table 12.

The goal of this evaluation is to understand the cost of `crash-sync-safety` relative to only providing `crash-consistency`, and whether providing `crash-sync-safety` provides any performance benefits compared to simply providing `crash-consistency`. We see that `crash-sync-safety` is a useful implementation property, as it can lower the cost of `crash-consistency` by scaling applications to multiple threads. For example, in the vacation benchmark achieving crash-sync-safety for 8 threads improves performance by $0.7\times$ compared to non-crash consistent single-thread execution, despite the crash-sync-safety property due to ccSTMs scalability. In contrast, the undo log causes a slowdown of $0.85\times$ to achieve crash consistency only (for a single-thread execution). In this section, we breakdown the costs of crash-consistency and characterize single-thread applications.

Summary of crash-consistency results. We find that the persistence domain plays a crucial role in choosing the best `crash-consistency` method. The same mechanisms have very different behavior and performance characteristics on systems with transient caches versus systems with persistent caches. In particular, `HTM` is an interesting case-study. For systems with persistent caches, `HTM` guarantees `crash-consistency` out of the box, with no architectural changes, while for transient caches, `HTM` needs architectural changes to ensure `crash-consistency`. In both cases, `HTM` also provides correct `synchronization`, and incurs the associated costs, although all applications we consider in this section are single-threaded and do not require `synchronization`.

From our empirical results, we can draw the following conclusions: (1a) In systems with transient caches, HTM benefits crash-consistency, despite its synchronization costs and required architectural changes. This is because, the HTM based crash-consistency systems are able to reduce the number of expensive cache line flush and fence instructions used in pure software undo/redo logging techniques. (1b) The choice between undo and redo logging vastly depends on the application characteristics and the size of the read and write sets of the transactions. (2a) In systems with persistent caches, the HTM benefit for crash-consistency is reduced, as software logging mechanisms do not require expensive flush and fence instructions anymore. In this case, the HTM's synchronization overheads become apparent. (2b) undo logging is the best choice for ensuring crash-consistency when caches are persistent, since redo logs still suffer from read-indirection overheads.

Overall, persistent caches provide a significant performance benefit, as the overhead of crash-consistency on average over seq is only 1% for the best method (undo), compared to 6% for best method when caches are transient (HTM+redo).

5.7.1 Transient CPU caches

We compare the performance of the various crash-consistency mechanisms on systems with transient CPU caches (§5.2.2), on both real hardware and our architectural simulator (§5.5).

Real. In this experiment, we evaluate all crash-consistency mechanisms on the real hardware using a server with support for TSX and we show the results in Figure 25. As expected, all crash-consistency mechanisms increase execution time compared to a non-crash-consistent baseline (seq). However, HTM+undo and HTM+redo significantly outperform their pure software counterparts (undo and redo), improving performance by as much as 98× and 97×, respectively. This is due to the HTM reducing the number of fences and read-indirection for the transactions that succeed. The only exception is Labyrinth, where

HTM+undo and HTM+redo perform similar to their software counterpart, due to more frequent aborts caused by large transaction sizes (Labyrinth). These results indicate the best performance that we can expect from a ccHTM on real hardware, as we approximate the performance using HTM+undo/redo that only provides crash-consistency on the fallback path, but not inside the hardware transaction. Therefore, these results measure the upper limit of ccHTM. For a more conservative estimate of ccHTM performance, we also evaluate it in the simulator (Fig. 26). The transaction success rate[1] varies from from 49% (Labyrinth) to 100% (Kmeans) - also shown in Figure 25.

HTM+undo and HTM+redo are on average 14% and 12%, respectively, of the ideal baseline, seq, showing that crash-consistency can be ensured at a small performance penalty using HTM. Although not needed for single-threaded applications, HTM also provides synchronization. To understand this additional overhead, we also evaluated HTM+seq, which ensures synchronization when transactions succeed, but not on the fallback path. HTM+seq, incurs overheads, ranging from 2.5% (intruder) to 13% (genome), even when no crash consistency guarantees are provided. These overheads are due to hardware bookkeeping to execute transactions and transaction aborts.

All pure software crash-consistency mechanisms (undo, redo and ccSTM) greatly increase execution times – as much as 41×, 89× and 33× respectively (for Kmeans). redo logging generally performs better or as well as undo logging for the workloads evaluated, but the results highly depend on the number of reads and writes in the transaction. undo suffers the overhead of flushing and fences for every write, while redo suffers the overhead of read indirection, proportional to the number of reads and the size of the log. ccSTM incurs the highest overhead across all workloads. We attribute this to the higher synchronization overheads of the ccSTM, in addition to the software logging overheads like undo and redo.

[1]TSX transactions are best-effort, so they might abort even for single-thread workloads, when there are no conflicts.

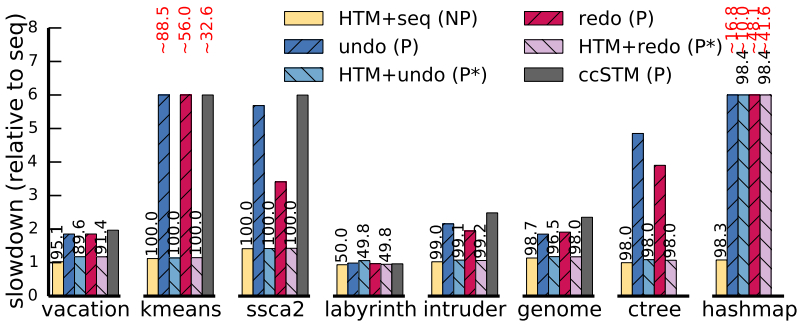

Figure 25: TSX-enabled hardware, with real PMEM and transient caches. We show transaction success rate for methods using HTM (values in black). We truncate large bars in Kmeans and hashmap (values in red). (P) crash-consistent; (P*) approximates a crash-consistent solution; (NP) not crash-consistent.

Simulated. In this experiment, we use the simulator to evaluate `ccHTM` for transient caches with the proper architectural changes. Figure 26 shows the results. The trends from the real hardware still largely hold here. `ccHTM-undo` and `ccHTM-redo` have lower overheads than their software counterparts by as much as 3.2× and 2.7× (Kmeans) respectively. And they both come within 1.2× of the ideal baseline, `seq`. Even with full support for `crash-consistency`, `ccHTM` outperforms other methods. The only exception is Labyrinth, where `ccHTM` suffers comparable overheads to its software counterparts due to frequent transaction aborts. However, the transaction abort rate is less in the simulator than on real hardware as the simulator models overflow and some unsupported instructions, but not all events that would cause a transaction to abort on real hardware. We attribute these differences to inaccuracies between the hardware implementation details within the simulator and proprietary commercial hardware. `ccHTM-seq` adds 28.62% overhead on average for all benchmarks compared to `seq`, highlighting that the architectural changes made to the `HTM` have fairly low impact.

5.7.2 Persistent CPU caches

In this experiment, we use our bare-metal testbed (§5.5) to emulate persistent caches (§5.2.2). We show the results in Figure 27. Unlike for transient caches, here `HTM+undo` and `HTM+redo` perform worse than `undo` and `redo` by at most 10%. When caches are persistent, `undo`

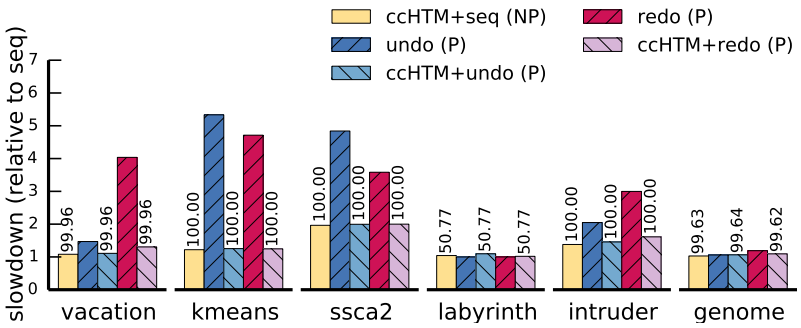

Figure 26: Simulation, transient caches. We show transaction success rate on top of the methods using HTM. For each method, we specify if it is crash-consistent (P) or not (NP).

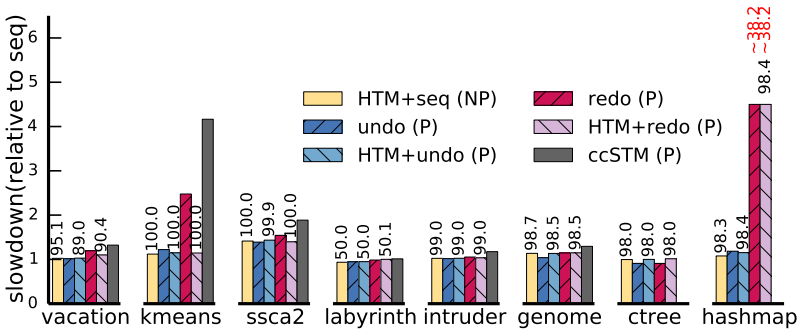

Figure 27: TSX-enabled hardware with real PMEM, emulating persistent caches. We show transaction success for methods using HTM (values in black). We truncate large bars in hashmap (values in red). (P) crash-consistent; (NP) not crash-consistent.

and `redo` logging techniques no longer have to use expensive cache line flush and fence instructions to ensure data consistency, while the `HTM` still has the overhead of ensuring `synchronization`. To quantify the overhead of the `HTM`, we measure `HTM+seq`, which is up to 40% slower than `seq`, while `undo` and `redo` are up to 40% and 37× slower. The `STM` has even higher `synchronization` overhead on average, being up to 3.1× slower than `seq`. Moreover, `undo` and `redo` perform similarly in most cases, except in the case of hashmap and Kmeans, where `undo` performs better than `redo`. Although both methods are faster because they don't require fences and flushes, `redo` still has the overhead of read indirection in certain workloads with many reads and writes. Moreover, an application can be tuned to use one technique or the other, which we show with hashmap as an example. Hashmap is tuned to use undo logging with PMDK library and thus perform better with undo logging.

5.8 Chapter summary

We introduced crash-sync-safety property which helps programmers with combining crash consistency and synchronization protocols for achieving ACID qualified durable,multi-threaded application programming on PMEM. we provide a comprehensive evaluation of the impact of combining existing crash-consistency and synchronization methods for achieving performant and correct PMEM transactional systems. We consider different hardware characteristics, in terms of support for hardware transactional memory (HTM) and the boundaries of the persistence domain (transient or persistent caches). By characterizing persistent transactional systems in terms of their properties, we make it possible to better understand the tradeoffs of different implementations and to arrive at better design choices for providing ACID guarantees. We use both real hardware with Intel Optane DC persistent memory and simulation to evaluate a persistent version of hardware transactional memory, a persistent version of software transactional memory, and undo/redo logging. Through our empirical study, we show two major factors that impact the cost of supporting persistence in transactional systems: the persistence domain (transient or persistent caches) and application characteristics, such as transaction size and parallelism.

CHAPTER VI

SUPPORTING RELIABLE PERSISTENT MEMORY PROGRAMMING

In this chapter, we further extend the discussion of memory native persistent programming on PMEM. The `crash-sync-safety` transactional (discussed in the previous chapter) programming primitive only offers failure-atomicity plus synchronization and lacks the reliability guarantees – an important trait for most enterprise applications. Node local persistent data is vulnerable to catastrophic media failures as we only maintain a single copy of the data. Though data reliability is a well-studied system problem, it has often been done in the context of either persistent block abstractions [76] or with volatile main memory [77] as the storage device. PMEM, supporting native memory persistent programming, changes traditional durable media programming abstractions and obvious trade-offs in providing reliable data storage. We discuss system software methods to support enterprise applications with durable and reliable native memory backends.

6.1 Introduction

Byte addressable persistence of PMEM encourages the end-user applications to jettison the traditional block I/O based system software intermediaries in favor of native memory persistence. However, it still leaves the worrying question of availability (when persistent media fails) and the right software interface to persistent memory that adequately hides the complexities of failure atomic PM programming.

Combining the much faster new persistent devices with existing replication protocols [78] shifts much of the end-to-end bottlenecks into the data transport and copying overheads of the replication stack. It also raises the question of how to couple the execution of the

persistence and replication engines, while providing for concurrent operations, needed for performance, and maintaining order, required for correctness. This is non-trivial because of the sequential nature of replicating a log of updates is at odds with the concurrent updates needed to maximize the performance that can be achieved on the new persistent media.

In response to these challenges, we present `Blizzard`– a fault-tolerant, PMEM optimized persistent programming runtime. `Blizzard` is a software stack that lets programmers build sophisticated, persistent applications as a service with only modest software modification requirements. Persistent data structures are exposed to the client application through an RPC interface, while `Blizzard` ensures the performance and correctness of the data structures' access, durability, and fault-tolerance operations.

In this chapter;

1. We design a PMEM-specialized replication stack that addresses the challenges of integrating replication, persistence and concurrency when combining PMEM-based persistence with high-speed networking.

2. We provide the implementation of the `Blizzard` system for commodity Ethernet networks with DPDK high-speed packet delivery, and Intel Optane DC PMEM, the first-generation of directly-attached persistent memories.

3. Experimental evaluation of `Blizzard` with different applications demonstrates its generality and flexibility for creating different persistent and fault-tolerant data structures that can be further combined as needed by sophisticated applications, while delivering to applications performance benefits and stronger availability guarantees.

6.2 Need for fault-tolerant, persistent memory

Persistent memory with correct system software primitives (§5) supports fast and durable application state maintenance using direct CPU load/stores. However, PMEMs integrated in commodity servers only provide limited reliability guarantees for persistent data in the

form of software RAID among PMEM modules, erasure codes, etc., but fail to protect data against catastrophic node or persistent media failures (hard-failures). Fault-tolerant persistent application state that survives hard-failures is a must for highly available enterprise applications.

The design of system software for truly fault-tolerant persistent memory programming is not trivial. We identify two main components in such a design: **Persistent memory programming layer** – that stores application specific persistent data, and **Replication layer** – that supports fault-tolerance by synchronizing persistent data between multiple server-replicas. Integrating these two components to form a high-throughput, fault-tolerant PMEM programming runtime is challenging, as the system software techniques commonly used to optimize each of these two components individually, often conflict with each other. For example, state machine replication protocols such as Paxos and RAFT [79, 78] rely on *serial log-commits* to preserve consistent replica state among distributed nodes, while high throughput persistent programming involves concurrent data-updates to offset some of the crash-consistent write-ahead-logging (WAL) costs (§5). Naive coupling of these two components will compromise performance (e.g., turn-off concurrent updates), or, worse yet, will lead to incorrect persistent data states.

This thesis presents Blizzard, a system service that supports truly fault-tolerant persistent memory programming. Blizzard takes care of availability and provides the client with access to the persistent state via an RPC interface. The approach is consistent with modern microservice-based datacenter stacks. To demonstrate and evaluate Blizzard, we will provide a small (but growing) set of persistent implementations of popular data structures that will allow performant implementations of a wide array of real-world applications.

The underlying innovation that makes `Blizzard` possible is exploiting direct access to persistent memory from both the CPU and the (commodity) NIC to provide efficient zero-copy replication of RPC calls. We build on this by providing a library of persistent data

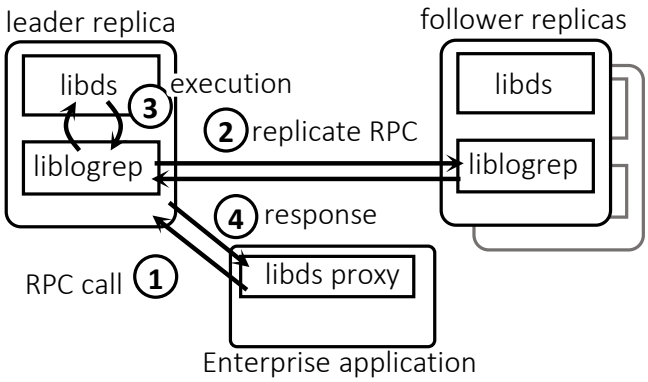

Figure 28: `Blizzard` consists of three main components. `libds`, `liblogrep` and Execution protocols that couple former two.

structures with a recipe for concurrency that works well with replication. Concretely, Blizzard provides an arbitrary number of logical ”channels” to named data structures held in persistent memory. Operations that do not commute go to the same channel while services that commute go to different channels. Blizzard enforces sequential execution on the same channel.

We next describe each of our design choices and their rationale in-detail.

6.3 Overview

We propose `Blizzard`– a fault-tolerant, persistent memory programming runtime with native support for in-memory durable data-structures. `Blizzard` achieves low-latency, high-throughput replication of in-memory state through use of an PMEM-specialized user-level replication stack. Performance in the replication path is further enabled by maximizing concurrency, while continuing to maintain application-specific ordering and correctness guarantees. The outcome is that `Blizzard` provides for reliable and persistent in-memory data-structures, capable of replacing existing enterprise application backends while providing improved performance, reliability and additional functionality.

`Blizzard` has two key components, `libds` and `liblogrep` (see Figure 28). `libds` implements a growing number of persistent memory data structures modeled after familiar C++ STL library counterparts, and `liblogrep` handles their persistent state replication

among node replicas. Below, we briefly summarize `libds`, `liblogrep` and the coupling protocols between them, and provide their detailed description in the subsequent sections (§6.4-§6.6).

- We design and develop `libds`, the application developer facing component of `Blizzard`. `libds` supports a rich set of commonly used reliable, persistent memory data-structures (e.g. maps, queues, etc). We implement `libds` data-structures using PMDK [21] – a popular persistent memory programming library. PMDK supports persistent memory programming primitives such as persistent memory allocators and durable transactions. The operations to create and manipulate each of the data-structures are exposed as a network call and made available to client-side applications via associated data-structure proxy. `libds` allows these data-structures to be combined to form complex backend data models required by real enterprise applications. At the heart of `libds` functionality, is the core `Blizzard` programming APIs supporting RPC style persistent memory programming over a network. We discuss `Blizzard` core API in §6.4.

- We design and develop `liblogrep`, an PMEM-aware, fast log-replication runtime that extends the RAFT log-replication protocol. `liblogrep` log-replications are durable and replicate `libds`'s operations across replica nodes. `liblogrep` carefully integrates userspace-networking and byte-addressabilty of PMEMs to realize end-to-end data zero-copy and batching, and to achieve low latency/high-throughput operation replication.

- `libds` and `liblogrep` are combined in the `Blizzard` execution layer in a manner that retains the zero-copy benefits while ensuring correct end-to-end execution of incoming data-structure operations, both within a single node (crash-consistency semantics) and across node replicas (distributed data consistency). Furthermore, for increased system throughput, the execution layer integrates operations scheduling

protocols allow concurrent execution of operations on replica nodes without compromising distributed consistency.

6.4 `Blizzard` Interface

The core operational model in Blizzard is a client-server one: users write services that receive remote procedure calls from clients, lookup and manipulate persistent state and then return a response. In order to provide maximum flexibility to the user, we treat the actual RPC call as a binary blob.

```
1  // Client side API
2  Status MakeUpdateRPC(const string& request, string* response);
3  Status MakeReadRPC(const string& request, string* response);
4
5  // Server side API
6  class Lock() {
7   public:
8    virtual void ReleaseLock() = 0;  // Override to release your
    lock
9  }
10 void HandleRPC(const string& request, vector<Lock>*
    delayed_locks, string* response);
11 bool Commutes(const string& requestA, const string& requestB);
```

Figure 29: `Blizzard` API

The listing above shows the Blizzard API available to programmers - we elide some setup details for reasons of space and focus on the core APIs. On the client side, the programmer can send an RPC call as a binary blob. Blizzard takes care of service discovery (primarily, locating the RAFT leader) and sends the server the RPC call. It returns the response as another binary blob and a Status object detailing whether the RPC call could be made successfully. Application level errors, if any, are encoded in the return blob by the application. We distinguish between read and update RPCs as separate calls. This is because reads are not replicated but updates need to be.

The server side API encapsulated in `HandleRPC` is implemented by the programmer

and executed as a callback by Blizzard. It contains the received request blob and expects a response blob to be returned after execution. For update RPC calls, Blizzard guarantees that the `HandleRPC` call is executed *after* the call has been successfully replicated on the RAFT log (Section 6.5). The `HandleRPC` call is executed in the context of a persistent memory transaction (Section 6.6). Any updates to persistent memory are only committed *after* `HandleRPC` returns to the Blizzard runtime.

We expect concurrent invocations of `HandleRPC` to access a shared (persistent) memory data structure and to do so in a thread-safe manner. The Blizzard `libds` provides some thread-safe data structures along with their client side data-structure proxies, the programmer is also free to roll their own using the PMDK library. In all cases data race safety enforcement is the responsibility of the programmer using their favorite lock implementation, which they are free to place in volatile memory. The only requirement is that they wrap these locks in an object derived from `Lock` in the Blizzard API and *do not* release them when executing the RPC callback. Instead, these must be returned in the vector object provided in the API. Blizzard releases all the locks *after* updates to persistent memory have been committed (we discuss the reasons for this in Section 6.6).

Finally, we expect the programmer to implement a callback to help determine the commutativity of various RPC calls. A programmer declares two RPC calls as commutative if either order of execution leads to the same result for both the calls and therefore, they do not care if Blizzard executes them in different orders on different replicas - for example, increments to a counter. Commutative RPC calls are executed concurrently by Blizzard allowing better performance than simply following the sequential execution specified by the replication log.

To make the server side API more concrete, the following listing provides an abbreviated benchmark from Lobsters [80] that we use in this paper. It maintains a hash table, mapping news story identifiers to vote counts. The data structure is provided by Blizzard

```
1  blizzard::map<string, int> votes;
2  pthread_mutex_lock big_lock; // in volatile memory!
3
4  class MyLockWrapper : blizzard::Lock {
5   public:
6     MyLockWrapper(pthread_mutex_lock* lock)
7      : saved_lock_(lock) {}
8     virtual void ReleaseLock() {
9       pthread_mutex_unlock(saved_lock_);
10    }
11  private:
12   pthread_mutex_lock* saved_lock_;
13 }
14
15 void HandleRPC(const string& request, vector<Lock>*
    delayed_locks, string* response) {
16   // Lock everything and stash away the lock for blizzard
17   pthread_mutex_lock(&big_lock);
18   delayed_locks->push_back(MyLockWrapper(&big_lock));
19   votes[StoryId(request)]++;  // Blizzard auto-undo logs the
    hash bucket!
20 }
21
22 bool Commutes(const string& requestA, const string& requestB) {
23   // Allow vote increments for a story to commute.
24   return true;
25 }
```

Figure 30: `Blizzard` sample code for retrieving top K voted entries.

and for this example we assume it is *not* thread-safe, requiring the programmer to implement their own locking. The RPC call handler adds a vote to a story.

The most interesting aspect of this example is the decision (by the programmer) to declare all vote increments to commute. Viewed from the perspective of a single story, increments and reads of the vote count are serialized. From the perspective of multiple stories however increments are not serialized as the updates to different stories are executed in different orders on different replicas. This reflects the fact that Blizzard provides strong consistency in terms of state machine replication in the underlying layers but allows programmers to relax that ordering for better concurrency. As we show in the following

sections, we address the performance of RAFT replication while maintaining the serial order of log updates, thereby providing a strongly consistent and performant substrate for programmers to build their persistent memory applications as they see fit.

6.5 Replication

Blizzard uses RAFT [78] to replicate a durable log of updates to persistent memory data structures. Replication is a necessary in Blizzard - it is critical to ensure in-memory data structures are truly available even when the underlying persistent memory fails or the machine goes down. However, log replication is a synchronous operation. The latency components of accessing durable storage and network hops to replicas add to the latency of operation completion. We note that although various pieces of work that have attempted to improve network overheads for replication protocols [81, 77, 82] *they have examined replication without persisting any state*. This is not an accident – flash storage comes at a significant latency and throughput cost compared to network performance. In contrast, Blizzard is designed ground up for persistent memory that is at least an order of magnitude faster than flash. This therefore puts us in a position to focus on the network component of a fully functional replication stack that *includes* persistence.

A key building block for Blizzard is userspace network access. RAFT (as well as most other replication protocols) are already designed to work on unreliable networks. Therefore, it makes sense to jettison stream oriented reliable delivery by the kernel TCP/IP stack in return for significant gains in latency to the wire. We use the Data Plane Development Kit (DPDK [83]) for fast access to the network from userspace. For our specific setup, this leads to a 3X reduction in latency for a single hop on the network from 28us down to 8us. Although this restricts Blizzard to operate in a single Data Center environment, where the Ethernet header is sufficient for routing from source to destination and networks are largely reliable, we believe that this is an acceptable tradeoff. We exploit the direct addressability of persistent memory to build a high performance replication stack using two simple

principles: zero copy and batching.

Copying log entries in various parts of the replication state machine - from receiving the client request to sending out copies to replicas - is expensive. This is even more so since persistent memory is still around an order of magnitude slower than volatile random access memory. The fact that persistent memory allows one data structure to simply point to another (rather than indirecting through a block address on flash) leads to an efficient solution to this problem in Blizzard. Figure 31 shows how RAFT log entries are organized in relation to DPDK's memory buffers. A DPDK memory buffer holding an incoming request (complete Ethernet frame) is placed in an aligned block of memory together with external metadata pointing to the start and finish of the block - we remind the reader that this is all persistent memory. We keep the client request in its DPDK memory buffer for its lifetime - spanning replication and execution.

To start with, the leader prepends a RAFT control block (with information such as term and index) to the buffer adjusting the external metadata to compensate. The leader is now ready to replicate the request. We exploit the fact that DPDK allows multiple memory buffers to be chained together. To do so it simply creates an Ethernet header for *each replica* and chains the same log entry packet to each of them. It hands off all the headers to DPDK. The NIC then does the heavy lifting of assembling the Ethernet frames and sending them out. We underline that this is only possible because the logs are not in block storage and persistent memory is accessible from all connected agents, including I/O devices, in the system. In contrast, directly accessing block storage from the NIC involves serious complications [84] and this design illustrates how persistent memory can simplify the design of distributed system primitives that need persistence.

Although this design lifts most of the load off the CPU, the RAFT consensus protocol still represents an overhead for each log entry. More pertinently, this overhead is sequential, since each log entry needs to be processed before moving on to the next one. A simple way to further improve performance under load was therefore to batch process log entries.

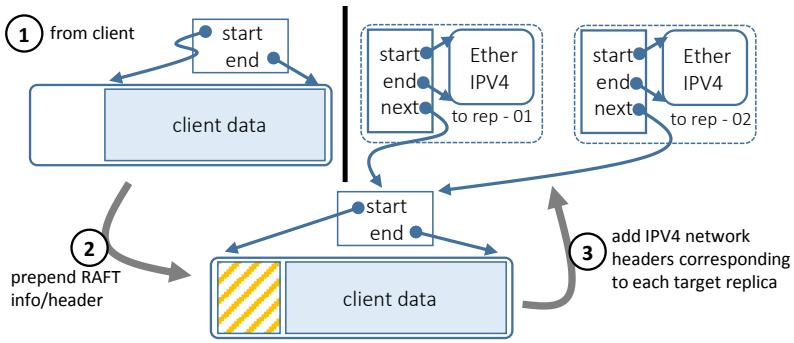

Figure 31: RAFT log entry in Blizzard

DPDK already provides an efficient vector interface to receive multiple packets waiting in the NIC queue. We chain these packet buffers together in userspace and treat them as a single RAFT log entry. This effectively amortizes the CPU cost of running the RAFT protocol state machine over multiple log entries.

Blizzard's RPC layer, together with the totally ordered semantics of replication with RAFT, means that we provide serializability in terms of the distributed consistency model, if we execute RPC operations in the RAFT log order and read operations as soon as they are received at the leader. Although, we do not replicate reads, the RPC layer always directs reads to the leader replica and thus, we provide read your own writes consistency in addition to serializability. We *do not* provide linearizability as that would require us to replicate reads to ensure that a leader does not become partitioned without realizing it and responds to reads without taking into account concurrent writes in the majority quorum. We believe serializability with read your own writes consistency is an adequately strong distributed consensus model for programmers to be largely oblivious to replication under the hood.

6.6 Execution Layer

Blizzard's execution layer aims to concurrently execute *committed* operations in RAFT's execution logs. We depend on the application programmer to specify commutativity between operations (Section 6.4). The execution layer couples to the replication layer via a

set of queues to receive operations on and uses flags in the RAFT log entries to track and update the state of each operation – replicating, replicated (or replication failed), executing and complete. The most complex part of Blizzard's execution layer is the scheduler that aims to schedule ready operations as soon as possible, while respecting commutativity. The actual execution leverages PMDK's persistent memory transaction library to enforce failure atomicity. We discuss each of these components below. Finally, we also discuss the implications of declaring operations as commutative and how the programmer can control departure from serial execution order for better performance.

6.6.1 Coupling

Figure 32 illustrates the overall design and interfaces between Blizzard's replication and execution layers. Every read and write operation received by the replication layer, is added to to a queue (`Q`) of operations, implemented as a persistent circular log. Each entry in the circular log is itself a pointer to the actual DPDK memory buffer holding the RAFT log entry. Each of the RAFT log entries include a set of flags read from and written to by both the execution and replication layers to allow them to communicate the state of the operations. Most importantly, these flags are persistent and survive restarts, forming the foundation for recovery.

All currently executing or ready to execute operations (a subset of `Q`) are a set (`E`), maintained in volatile memory. A scheduler picks operations from `Q` and adds them to `E` when ready to execute. Executor threads pick operations from `E` to execute and update the flags in the RAFT log when execution is complete. As part of this post completion operation, the execution thread marks the operation for garbage collection by setting the `gc_flag`. The replication sub-system uses this information to decide when to garbage collect the RAFT log and move the tail of the persistent circular log forward.

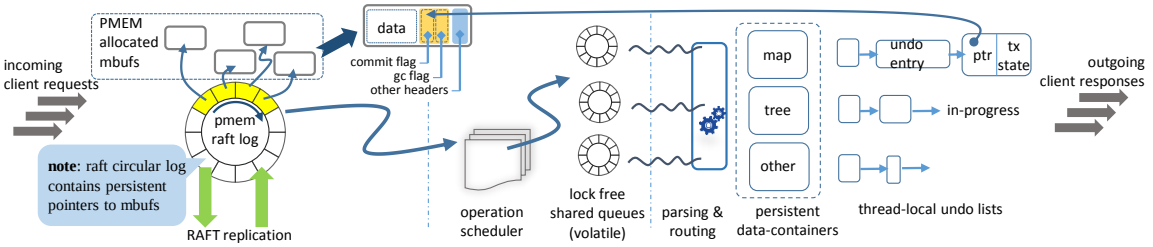

Figure 32: Blizzard - Coupling replication to execution

6.6.2 Scheduling

The scheduler runs as part of the continuous event loop in Blizzard, executing the scheduling algorithm shown below.

Algorithm 1: `Blizzard` operations scheduler algorithm.

input: 1. queue Q of updates and reads.
2. Set E of operations that are ready-to-execute/executing

```
1 repeat
2     if Q.head().state == FAILED_REPLICATION then
3         Q.dequeue()
4     else if Q.head().state != REPLICATING and Q.head() commutes with all ops
        in E then
5         op = Q.dequeue()
6         E.insert(op)
7 until server-shutdown;
```

The scheduler considers for execution operations at the head of `Q`, either immediately – for reads, or only once they are successfully replicated – for updates. The scheduler checks each operation against all currently executing operations in the set `E`. If it commutes with all operations in `E`, the operation is added to `E` for execution. Operations at the head of `Q` that have failed replication (perhaps due to a RAFT leadership change after the operation was received) are removed from consideration by the scheduler.

6.6.3 Execution

The execution of operations in Blizzard is done by dedicated executor threads. Each thread repeatedly pick up (with appropriate synchronization) an operations from the set `E`, executes

it, and after the execution is completed and any of its effects persisted to memory, it is removed from E. The execution follows the steps detailed below:

Algorithm 2: `Blizzard` Execution algorithm.

```
   input: o = operation to execute, E = set of executing/ready operations
 1 if o.state != COMPLETED then
 2     BEGIN FAILURE_ATOMIC_TX
 3     remove o from E
 4     lockset = []
 5     HandleRPC(o.request, lockset, o.response)
 6     o.state = COMPLETED
 7     END FAILURE_ATOMIC_TX
 8     foreach l in lockset do
 9         l.ReleaseLock()
10 remove o from E
11 Send o.response to client
```

We note that although the scheduler ensures only commutative operations can execute simultaneously, that does not mean those operations will not conflict when accessing memory. For example, increments to a counter are commutative but one still needs to synchronize on access to the counter to avoid two updates reading the same initial value of the counter. In addition, we require a total order on simultaneously executing concurrent updates to make recovery possible. To see why this is so, consider two increments of the same counter. These are commutative. Assume one increments the counter from 0 to 1 and releases the lock. The second now increments the counter from 1 to 2 and then commits its changes to persistent memory. On a failure, we need to replay the first increment, which now reads the value 2 from the counter (instead of 0). Therefore we need to ensure commutative operations serialize with each other as a whole when accessing the same memory location. We enforce this via delayed release of locks – a fairly standard technique borrowed from databases. The execution algorithm ensures that any locks acquired during execution (by user code) are released only after execution and persistence are complete.

The replication and execution layers cooperate to ensure that the states for an operation move as per the state machine diagram shown in Figure 33.

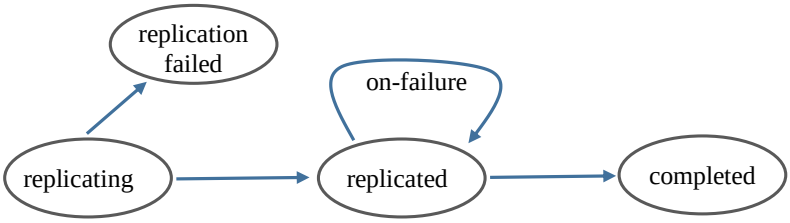

Figure 33: State machine diagram of an operation in Blizzard

We draw particular attention to recovery. The persistent circular log and state flags in the RAFT log entry form the foundation for recovery. We process all undo logs and then start the scheduler. The delayed lock release ensures that any operation that had completed does not see any change to its input data. If an operation begins execution but fails before executing, its persistent state flag remains set at `REPLICATED` when the system restarts. Any persistent changes made by the previous execution are automatically undone by PMDK. It then proceeds as usual through the current attempt till completion. On the other hand, if an operation finishes execution and manages to move its persistent state flag to `COMPLETED`, we do not execute it again by checking for this condition, thereby ensuring that operations are executed exactly once with respect to changes to persistent memory.

6.6.4 Commutativity

We now consider how commutativity impacts the distributed consistency model. The precise definition of commutativity that we provide to Blizzard programmers is: *Two operations commute if the result returned by each of them, when executing one after the other, does not depend on the order of execution*. Note that this relation is transitive. When no two operations are defined as commutative, execution occurs in RAFT log order and therefore we provide serializability with read-after-write consistency with respect to the data structure operations. If two operations are declared to commute, Blizzard might execute them in different orders on different replicas but - by definition - this cannot change their results.

Commutativity can be a powerful tool to extract parallelism from the sequential order specified in the log. As an example of allowing some commutativity, consider a single

container in persistent memory with a dictionary (implemented as a persistent hash table or tree) interface. Most such APIs (e.g., in C++ STL containers) disallow operations to multiple keys. Therefore a natural setting for commutativity is to allow operations (reads or writes) to different keys to commute, since reads by clients cannot reveal out of order application of operations to different keys at different replicas. In a such situation, the programmer can set `Commutes` to return true if and only if the operations are made to different keys. The result is also serializability with read your own writes consistency when the data structure API is *restricted* to a single key.

As an example of a more complex commutativity specification consider an example of a graph stored in persistent memory, as an adjacency list: a map of vertices to a list of neighboring vertices. Adding or deleting edges can be tricky due to the need to update both source and destination vertices. We need to ensure that reads see a consistent state of the graph: reading attributes of an edge specified as (u,v) should succeed and return the same result regardless of whether we lookup vertex u or v to retrieve edge information. An intuitive setup here is to allow edge changes to commute if they do not touch the same vertex: $Commutes((u,v),(x,y))$ should return true if and only if $\{u,v\} \cap \{x,y\} = \emptyset$.

We show in the evaluation that judicious settings of commutativity allows more concurrency to be extracted from the single serial order in the RAFT log and therefore better performance. We note that the assertion that APIs that allow more calls to commute lead to more concurrency is in fact a *general* notion as shown in COMMUTER [85]. A future direction of work for us is to use automated tools such as commuter to determine the commutativity of API calls, rather than requiring programmers to specify it themselves. Manual specification of commutativity exposes parallelism and better performance but also leaves the potential for specification errors to lead to divergent replica states that can be hard to debug.

Table 14: Hardware/software configurations of the `Blizzard` testbed

Compute	Intel Xeon Cascadelake, 96 cores @ 1 GHz over 2 NUMA sockets. Running Fedora Linux.
Memory	375 GB of DRAM memory and 756 GB of direct accessible PMEM memory side by side to each other
Flash	420 GB Intel 520 series
Network	Commodity Intel 10-Gigabit Ethernet network cards with Intel DPDK userspace network stack.

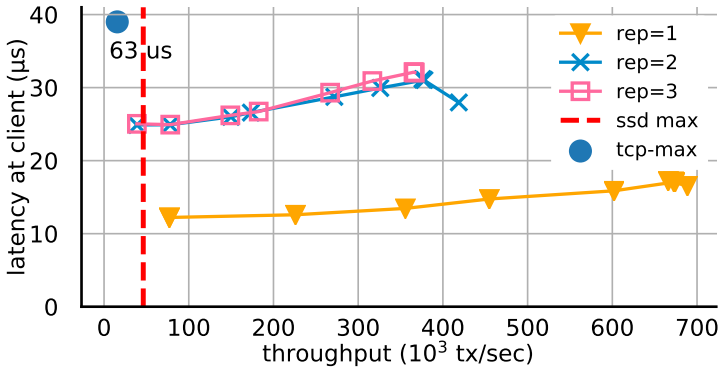

Figure 34: Blizzard's performance characteristics for an echo workload.

6.7 Evaluation

Setup: We evaluate `Blizzard` on a cluster of three identical servers (Table 14) and a set of client machines to issue RPC calls. The servers are connected to each other via a 10 GigE switch with jumbo frames enabled to facilitate maximum batching in `Blizzard`. Clients talk to servers via the same Intel DPDK network transport. Each server node is equipped with DRAM and PMEM memory modules that are accessible via load/store instructions. An `ext4` DAX file-system configured to use 2 MB hugepages [86] manages the PMEM address space. Unless otherwise mentioned, we allocate incoming/outgoing messages (DPDK mbufs), and RAFT log entries, persistent data-structures and their undo logs entries from PMEM. Additionally, we also equip the server nodes with a flash drive to contrast performance with PMEM.

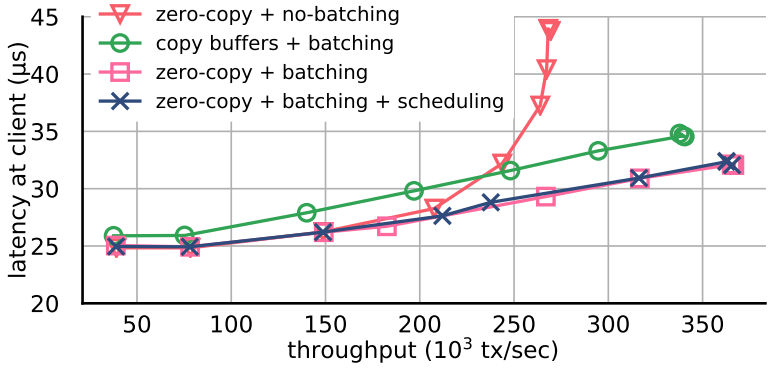

Figure 35: Importance of `Blizzard`'s performance optimizations using zero-copy and batching

6.7.1 Replication

We begin by evaluating `Blizzard`'s replication performance using a focused microbenchmark where the execution of the log entry is a no-op: the leader returns a response to the client as soon as the log entry has been replicated to a majority of the quorum of server nodes. `Blizzard` is configured to batch 32 log entries into a single RAFT log entry for efficiency. To put the results in perspective, we also measure the same with a regular network transport by replacing our userspace networking stack with TCP/IP. Furthermore, we also plot the theoretical peak performance if we were to batch write 32 log entries at a time to flash storage (32*1/flash latency throughput at flash latency).

As the results in Figure 34 demonstrate, the presence of PMEM removes storage bottlenecks and allows replication to be unhindered by the cost of persistence. On the other hand, this necessitates an optimized implementation to reduce network overheads so as to prevent the network stack from, in turn, becoming a bottleneck. The result is a system that is able to achieve a raw replication rate of $\sim 365K$ log entries a second (3 ways with full persistence).

Next, we evaluate the performance gains and overheads of the proposed `Blizzard`'s optimizations using the same no-op log entry replication (3 way replication with full persistence) microbenchmark. We use two base line numbers, 1) `Blizzard` replication without

batching optimization (no-batching), where we process one operation at a time during replication 2) Blizzard replication without zero-copy optimization (copy-buffers), where we make copies of incoming RPC payload, both during RAFT log-appends and batch preparation during network multicast. We compare these baselines against a Blizzard instance with both zero-copy and batching optimizations in Figure 35.

Fully optimized Blizzard handles 40% more traffic in comparison to Blizzard without batching optimization. We attribute the performance gains of batching optimization to; 1) increased memory parallelism during RAFT log appends. – a batched raft log entry append (for 32 ops) requires only a single store fence instruction after subsequent CPU cache line flushes, whereas no-batching version requires 32 of them. 2) reduced network multicast cost, as the RAFT metadata appends and control path operations are amortized over the number of batched operations. Similarly, without zero-copy optimization Blizzard's per operation latency increases by, 7% upto 34.5 us, across the throughput range, due to extra memory management steps – allocation,data-copy and free. Combined together, zero-copy and batching optimizations push Blizzard's maximum throughput to 365K ops/sec while keeping the per operation latency down at 32 us.

While Blizzard's zero-copy and batching optimizations have improved replication throughput, they also have shifted the system bottleneck from the replication layer, back in to the persistent data-structures. Therefore Blizzard supports a commute-scheduler that enable concurrent operation execution on data-structures without compromising correctness. We use the microbenchmark to measure the overheads of our commutativity respecting scheduler. We add in the scheduler but apply no constraints by allowing all operations to commute with each other - thus not changing the behavior but adding the overhead of the scheduling checks. The result is close to the the microbenchmark with no scheduler, demonstrating that the dominant cost in checking commutativity is on the side of the callback provided by the user, rather than the scheduler implementation.

Finally, we show how Blizzard handles replica failures to provide applications with

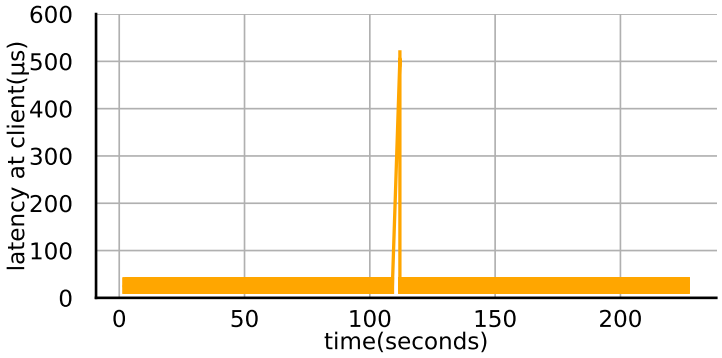

Figure 36: `Blizzard` failover performance in a 3 node replica cluster.

crucial availability guarantees beyond raw performance improvements. In Figure 36 we show the failover timeline of a `Blizzard` cluster with 3 nodes under echo/no-op workload, where we kill the leader replica in the midway. After time out based failed leader detection, `Blizzard` client probes the other replicas for new leader. For a failure detection timeout of 12ms, `Blizzard` fails over to new leader within 24 ms in the worst case for a 3 node replica cluster.

6.7.2 Key Value store

We now consider a general purpose application that is often used as a persistence layer in more complex systems - a persistent key-value store. Traditional persistent KV stores like RocksDB, often use complex data structures such as LSM trees [87] to compensate for the fact that block storage does not allow for easy random access. Could we leverage the simplicity of data structures in PMEM to build a performant replicated key value store? The in-memory data-structure realization for the same abstraction is a simple hash-map. Given the capabilities provided by `Blizzard`, we ask the following question; Is it feasible to implement a truly- persistent hash-map using `Blizzard` âĂŹs programming model?

We implement a hashmap based key-value store that supports point queries. We port PMDK's concurrent hashmap implementation as a `Blizzard` data-structure by extending the hashmap provided in PMDK with `Blizzard`'s crash-consistent update protocol and

a commute handler that honors a serialized read-your-own writes consistency model, as outlined in §6.6.4. Our implementation supports arbitrary strings of characters as both keys and values. The porting required only an additional 96 lines of C++ code.

We select RocksDB backed by PMEM (to give it the advantage of faster persistent media) as our baseline to compare `Blizzard`'s persistent hashmap against. We run write-ahead-logging enabled (for crash-consistency) RocksDB on `Blizzard` with replication and commutative scheduling turned off, thus Blizzard only serves as an RPC transport for RocksDB. We use 8 byte key/value strings for both RocksDB and the `Blizzard` hashmap-based key-value store, and use a Facebook-like [88, 89] workload with 50% writes. We report the resulting latency and throughput numbers in Figure 37.

`Blizzard`'s replicated and crash-consistent concurrent hashmap-based key-value store outperforms RocksDB's (no-replication) peak throughput by an order of magnitude, merely due to not being constrained to use block interfaces and LSM trees. The former's throughput further improves by a factor of two, upto a throughput of 270K ops/sec, once we mark operations to different keys as commutable thereby removing the constraint of serial execution imposed by the RAFT log. Furthermore, we did not see a significant increase in operation latency of our hashmap, where it remained < 44 us across the throughput range. This result therefore underlines that PMEM decouples persistence from the block abstraction and `Blizzard` allows programmers to exploit this without being burdened by implementing their own crash consistency or replication.

6.7.3 Graphs

Next, we evaluate `Blizzard` in the context of graph databases – an important class of applications that presents unique durable data management challenges. Real world interactions between entities such as in friend-networks can be natively represented using graphs. The most natural representation of graphs for search and traversal uses pointers – a source of great difficulty with block interfaces such as those in front of disks and flash. This has

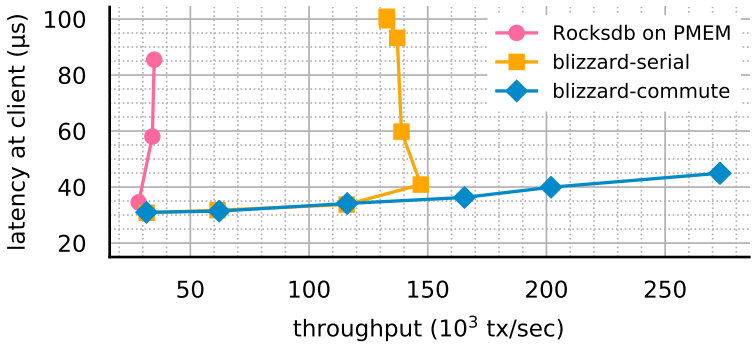

Figure 37: Blizzard's key-value map performance comparison (replicated and persistent) against similar software stacks

spawned a whole genre of research into batch processing of graphs from secondary storage [90, 91, 92].

PMEM, natively allows pointers and persistence to co-exist, thus presents a unique opportunity to solve this vexing problem. The key challenge is to do this without letting persistence or replication add programming complexity or eating unduly into performance. We examine whether `Blizzard` is up to this task. An adjacency-list is the natural data-structure for representing graphs on memory. We first implement a persistent adjacency list data-structure by putting together already available PMDK building blocks. Persistent list structures contain neighbor lists and a hashmap structure maps a vertex's `node_id` to corresponding `list_entry` of the row. Next, we extend the implemented graph-structure with `Blizzard`'s crash-consistency semantics. Finally, we implement the handler for parallelizing commutative operations. The implementation only took 110 lines of C++ code, as the bulk of the building blocks were already available as open source libraries.

We compare our implementation with LLAMA [93] graph processing system. LLAMA [93] models a graph using compressed sparse row (CSR) representation that supports good update performance with block devices. LLAMA treats each update as an immutable versioned object and aggregates them to form a consistent graph view during reads. LLAMA's delta based updates are specifically designed to handle frequent/streaming updates to graphs stored on durable storage. We run LLAMA on PMEM as a network service, using `Blizzard`'s

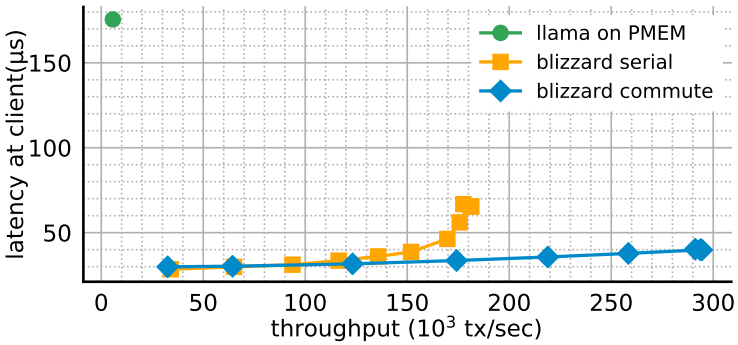

Figure 38: Twitter benchmark. Persistent graph use case

RPC transport.

We use the Twitter data-set [94] as our streaming graph benchmark. The data-set includes a subset (up to 15M nodes, 46M edges) of Twitter social network in the form of who-follows-who. We use half of the data-set to pre-load our graph database and use the rest of the updates to form a read/write (50/50) streaming workload similar to [95]. We read the out-degree for a given node, as our read operation in this workload. We report latency/throughput numbers for each of the competing systems in Figure 38.

The `Blizzard` graph representation based on a persistent adjacency list can handle up to 150Kops/sec, at <38 µs, all while providing full fault tolerance semantics. Our competing system, durable-LLAMA performs poorly at a peak throughput of mere 5K ops/sec and takes as much as 175 µs for the same workload. The poor LLAMA numbers surprised us, as we expected LLAMA with 1) no-replication and 2) PMEM as main memory, to have a performance 'edge' over the competing system. We learned that, LLAMA's copy-on-write (COW) updates are known to incur high overheads when the ingestion batch size is <10K updates, thus negatively impacting the system performance under frequent updates (we verified this with the LLAMA authors).

Finally, the parallalized adjacency-list representation by means of commutativity rules, further improve our graph operation throughput to 291K ops/sec, a $\sim 2\times$ improvement over a strictly serial execution schedule without significant increase in operation latency. Overall, our replicated and fault-tolerant adjacency-list based native graph representation

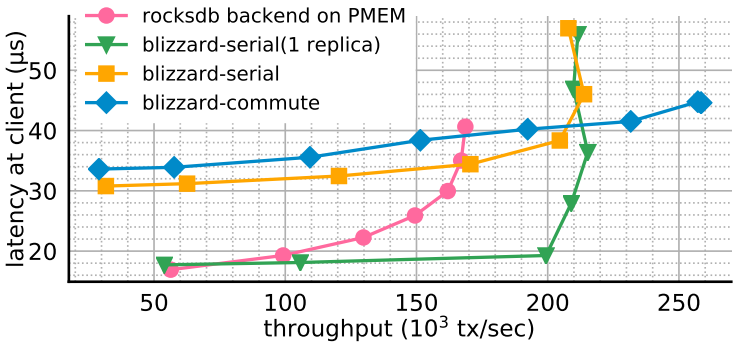

Figure 39: Noria's vote benchmark performance numbers against different storage backends.

outperforms the LLAMA graph engine by an order of magnitude ($\sim 58\times$), even without exotic data-structure level optimizations.

6.7.4 Lobsters

Finally, we use `Blizzard` to implement and evaluate a persistent data storage backend for a popular web-application, Lobsters [96]. Lobsters is a community based news aggregation site where users vote for submitted web-links. They display the top-K voted web-links on their home page. The original site manages web-links and their vote counts using a relational DB backend [80]. They model durable state using `article(article_id, web-link,...)` and `vote(article_id, vote_count)` relations. An article submit inserts a new entry into `article` relation and an upvote/downvote updates the `vote` relation. The top-K voted article list is maintained as application logic.

We use a persistent priority queue as the memory native data-structure to serve top-K requests. Such a data-structure is both intuitive and removes auxiliary book keeping in the form of application's runtime state as the data-structure itself maintains ordering with updates in logarithmic time. We use a max-heap to maintain the top-K voted stories and a min heap for the remaining stories. An upvote potentially causes the most voted story in the min-heap to move to the max-heap displacing the story with the minimum votes in the current top-k. A downvote can cause the opposite to happen. The hashmap maps

`article_id`s to min/max-heap entries as the incoming vote requests are indexed using `article_id`. For parallelism, we use a sharding scheme with multiple min-max heap pairs and shard the story keyspace across them. Determining the top-k becomes slightly more expensive due to the need to combine the results of the union of the min-heaps. We combine k entries from each sharded priority queue to form the final result.

We re-used a persistent hashmap implementation from [21] and implemented a persistent top-k priority queue using newly written min/max-heap code. Implementing the core persistent data-structure and operations took $\sim$ 600 lines of C++ code. We only allow update operations to different shards to commute with each other.

We compare `Blizzard`'s fault tolerant priority-queue data-structure performance against a RocksDB based Lobsters backend similar to [80], that is hosted on PMEM. Both `article_id` to `title` and `article_id` to `vote_count` mappings are encoded as key-value strings. The original Lobsters application backend of relational databases uses an additional relation and updates (at the cost of redundant data updates) to compute top-K. For a key-value store backend a top-K operation would be very costly as a join routine needs to be performed external to the data store (e.g. in application logic). Therefore, for the RocksDB backend we convert top-K requests to simple read operations similar to [80], thus pushing the RocksDB backend to its best case performance.

We run the Vote benchmark from [80]: a zipfian load generator that is modeled after actual website traffic. We pre-load our data-stores with 1M articles and run the experiment with 19/1, read/update traffic. The update operations consist of up-votes/down-votes of articles. During a read, the RocksDB based Lobster backend simply returns the article information for a given `article_id`, whereas the priority-queue based `Blizzard` backend returns top-K articles at a given instance. Therefore a top-K read request with the `Blizzard` implementation on average move k$\times$ more data than the RocksDB counterpart. We use K=8 over a 4 way sharded priority queue and issue one write operation for every 20 request in the experiment run. We record the throughput/latency numbers seen at the benchmark

client and report them in Figure 39.

The RocksDB powered Lobsters backend serves up to 160K ops/seconds with operation latency of <30 µs on average. Our persistent priority queue based `Blizzard` backend only incurs <55 µs on average, but manages only a peak throughput of $\sim 200K$ ops/sec. It is important to note that the RocksDB performance numbers benefit from the relatively simple read workloads (no top-K) and lack of fault tolerance (no-replication) over `Blizzard` in this experiment setup. The sharded and parallalized version of the same data-structure manages a maximum throughput of $\sim 257K$ ops/sec while keeping average operation latency <45 µs. Data structure sharding along with proper parallelization of incoming operations using commutativity helps `Blizzard` to handle 25% more traffic, while maintaining the same user APIs.

6.8 Chapter summary

We design and implement `Blizzard`– a replicated persistent memory runtime that supports fault tolerant, concurrent and persistent data-structure programming. `Blizzard` integrates userspace networking with byte addressable PMEM for a fast, persistent memory replication runtime. End user applications consume `Blizzard`'s persistent data structures using a familiar operations APIs, similar to C++ STL containers. Additionally `Blizzard` also implements replication and PMEM aware, crash-consistency and synchronization protocol enabling consistent and concurrent updates of persistent data-structures.

CHAPTER VII

RELATED WORK

The work presented in this dissertation makes contributions along several dimensions: high-performance streaming I/O, persistent memory programming, reliable distributed persistent services. In this chapter, we summarize the research most closely related to our work.

7.1 Applicaiton I/O in scientific computing

7.1.1 Application-driven multilevel checkpoints

application-driven checkpoints use explicit checkpoint interfaces to durably save application state and thus require changes to application code. As a result, only the state required to restart is checkpointed, and the application-specific knowledge provides flexibility to optimize and minimize C/R overheads. Application-driven HPC C/R mechanisms traditionally use a remotely located parallel file system (PFS) for checkpoint storage. Multilevel C/R [42] make use of locally available durable storage to perform frequent checkpoints, enabling the application to recover from soft/transient errors while alleviating the contention on the remote storage. Both `NVStream` and `PHX` system designs target application-driven, multilevel checkpoint use-cases.

7.1.2 I/O in HPC analytics pipelines

The main simulation scientific applications are often coupled with analytics applications, forming data workflows [26, 30]. The high data movement costs and increasing node-local processing power encourage in-situ analytics where simulation and analytics applications are co-run on the same physical machine. ADIOS [26] couples transport across simulation

and analytics by using files or volatile memory buffers. However, it lacks support for crash-consistent updates. Next, MPI-IO supports parallel I/O API when a parallel file-system backs destination files. While `NVStream` does not offer a file-system API, the thread-local log-structured memory in `NVStream` maximizes parallel I/O performance. Finally, TCASM [30] is a kernel-based system software component that supports streaming workflow data management based on a memory-mapped interface. It is designed for volatile memory and lacks versioning support apart from immediate-next semantic.

7.1.3 Persistent memory and HPC I/O

File API based HPC checkpoint and analytics I/O directly benefits from the PMEM optimized file-system implementations. PMFS [8] is a direct-access file system (DAX) that removes the page-cache layer between the persistent memory device and the file-system API. It uses undo logging for maintaining metadata consistency and copy-on-write for data consistency. BPFS [62], another file-system optimized for PMEM, uses copy-on-write to maintain its B+-tree based metadata structures for crash-consistency. NOVA [9], another PMEM-based file system, provides better performance over PMFS using a log-structured design. However, NOVA also suffers from system-calls, metadata updates, and garbage collection costs.

7.2 Crash-consistent persistent memory programming

New PMEM hardware with byte addressable loads/stores, can readily support pointer based native memory application structures. However, extending the same structures with pointer-based durability semantics is challenging as it involves handling crash-consistency semantics.

7.2.1 Storage transactions

Mnemosyne [11] proposes a redo logging scheme that encodes extra information in log-entries. The the protocol cuts back on the number of memory ordering instructions during

transaction commits and thus increase the hardware level memory parallelism during write-ahead-logging. DudeTM [97] works around the costly memory ordering and cache-line flushes during PMEM storage transactions using a shadow copy of the PMEM data in volatile memory. It keeps the PMEM's gold copy consistent by replaying a redo-log using a background task. Persistent, tree data-structures has the option of using copy-on-write protocols for maintaining crash-consistency. These protocols buffer updates in a partial copy of the tree structure before being atomically swapped into the original tree.

7.2.2 Synchronization and durabiliy

Synchronization protocols are fundamental to correct multi-threaded programming. Correctly synchronized, multi-threaded applications guarantee atomicity, consistency, and isolation (ACI) during program execution. We identify two main flavors of concurrency-control protocols, pessimistic and optimistic concurrency control (OCC). In pessimistic concurrency control protocols, acquire program locks before accessing shared data/state. The optimistic schemes use combination of sandbox data-execution environment and timestamp ordering to differ lock-handling until the end of critical section [69, 98, 99]. Multi-version concurrency-control (MVCC) schemes further extend [100, 101] the parallelism by supporting different isolation guarantees.

Supporting multi-threaded PMEM programming requires supporting durability in addition to atomicity, consistency, and isolation (ACID). Therefore it is natural to integrate PMEM support for existing multi-threaded applications by extending their concurrency-control protocols with PMEM aware crash-consistency semantics. However, the early efforts [10, 102] of integrating synchronization and persistence were specific to some system software domain. Atlas [103] formally defines and generalizes the mechanism for deriving a consistent program state using the synchronization semantics of application program structure. NVThreads [104] learns from Atlas and designs drop-in support for PMEM aware crash-consistency for multithreaded applications by extending Pthread locking.

7.2.3 Transactional programming

Transactional APIs have already been used as synchronization (e.g., transactional-memory) and crash-consistency (e.g., storage-transactions) primitives, therefore qualify as the unifying programming abstraction for PMEM aware ACID programming. Mnemosyne [11] and NV-Heaps[13] extend software transactional memory (STM) engines with transactional APIs to support PMEM persistence. The `ccSTM` that we evaluate closely follows the design proposed by Mnemosyne. PHTM [14] extends hardware transactional memory (HTM) with persistence using non-transactional stores and transparent flush semantics to ensure crash-consistency. PHTM was extended in PHyTM [15] by adding an STM in the fallback path. Both PHTM and PHyTM emulate logging inside the HTM region using regular load-/stores instead of their non-transactional stores and transparent flush support. Thus, their design affects the read/write set and capacity aborts. Also, PHTM and PHyTM provide only an approximation of their system performance using a TSX host, but no implementation of their proposed hardware extensions. NV-HTM [105] introduces HTM accelerated persistent memory transactions without changing the existing HTM hardware protocols. NV-HTM differs durable log-commit till HTM-end for correctness reasons, and thus misses out on overlapped durable log-writes.

7.2.4 Hardware support for PMEM

Intel added new instructions `clflushopt` and `clwb` for efficient transient cache-line flush [70]. Using these semantics and extending them, researchers have proposed persistency models [38, 106, 107, 108, 109] to reason about crash-consistency for PMEM. The industry trends suggest that persistent caches [110, 111, 112] may become prevalent. Recent work uses logging with persistent caches to provide durability guarantees [113, 114].

7.3 Reliable persistent memory

The work discussed in §7.2 enables efficient, convenient, and correct native memory programming on PMEM. However, these programming techniques neither support PMEM data reliability nor availability, as they are vulnerable to PMEM media failures.

7.3.1 Reliable systems design

Reliable and available systems are realized by replicating system state across machines in different failure domains. Supporting adequate distributed consistency among replicated data while maintaining excellent system performance is one of the key challenges in reliable system software design. Paxos [79], Viewstamp [115] replication and related family [116, 78] of protocols/services coordinate replica nodes to keep the replicated data in-sync. These protocols additionally integrate steps to handle membership changes/failures of the replicas. Applications directly use these services or use programming abstractions [117, 118] that rely on the former for reliable system design. Paxos-like replication services often follow strict ordering during data-replication and, therefore easily support serializable reads/writes – a very useful system property when building distributed applications using these services.

Other systems, choose to relax replication ordering for improved system performance, yet supports serializability by incorporating alternative system software ticks. CRDT like system abstractions [119] guarantee strong distributed consistency using restrictive API designs. CRDT commutative operations guarantee the convergence of distributed state without explicitly ordering operations among replicas. Tapir [120] supports fast, serializable, distributed transaction processing on inconsistently-replicated (IR) storage. Tapir combines OCC based concurrency control and specific application-level conflict resolution steps to achieve a strongly consistent replica state. COPS [121] tracks dependencies between data and introduce causal-consistency for scalable, replicated data-store design. COPS uses the extra dependency information associated with data for convergent conflict

handling among replicas.

Finally, systems such as Amazon Dynamo [122] further relax distributed consistency guarantees in favor of performance and availability. Aurora allows inconsistent replica state by design and resolves conflicts eventually by using application-specific conflict resolution routines.

7.3.2 Deterministic concurrency control

`Blizzard` uses an operations log (also known as logical log) with Paxos/RAFT protocol for strongly consistent data replication. Execution threads then apply these replicated operations on to the target data-structures. Concurrently executing a log of operations on to a properly synchronized target data-structure only guarantees an arbitrary chosen single serializable order among many such correct orderings. Such behavior is due to non-deterministic thread scheduling policies/mechanisms of OS kernels [123, 124] during operations execution. `Blizzard` replicas executing different serializable thread schedules may result in inconsistent state among replicas. Deterministic concurrency-control (DCC) protocols solve this challenge by explicitly controlling the thread schedules.

Data partitioning is the most straightforward DCC protocol. Partitioned data-structures isolate each executing thread to single partition and therefore support concurrent execution without affecting the data-consistency. H-Store [125] uses this system technique and partitions the database-store and assigns a single core for handling partition updates. However, cross-partition updates still need additional co-ordination in H-Store. Deterministic lock ordering algorithms [126, 127] grant locks to executing operations in the order that they appear in the operation log. However, the technique offers limited concurrency when the lock orderings of the operations are not known before their execution. Other proposals [128, 129] model dependencies/races between operations as a dependency graph. They then come up with partial thread schedules for concurrent executions while maintaining consistency properties. The relative costs of graph-generation and analyzing steps of the

generic thread scheduler are high in the context of `Blizzard`. Therefore we opted for commute-scheduler – a specialized version of the former technique.

7.3.3 Reliable PMEM programming

Recent work on PMEM has explored the feasibility of supporting reliable PMEM programming. Mojim [130] provides reliable, PM aware data storage by overloading `mmap/msync` system calls with synchronous replication support. Mojim uses an external co-ordinator and only supports strong replication up to two nodes (one mirror node) in its most common deployment model. AsymNVM [131] has a client-server model for PMEM programming similar to `Blizzard`. AsymNVM uses both value and operation logging for efficient log replication and features an LRU cache in the client for fast reads. It also uses an external co-ordinator for consistent state management. In contrast, `Blizzard` uses a unified replication log for both consensus and operations tracking – enabling a much simpler deployment than prior work.

7.3.4 Distributed shared memory on PMEM

Distributed shared memory (DSM) systems [31, 132, 133] pool main memory capacity across a machine cluster to form a one big (logical) shared memory compute engine. DSM systems support familiar memory load/store programming APIs , and therefore it is straight forward to port concurrent, shared-memory (node-local) applications into DSM. However, most DSMs use memory pages (4096 bytes) as the underlying data sharing unit between distributed nodes because of the hardware support (processor MMU) and data-sharing tradeoffs. Therefore, DSM programming requires careful placement/accesses of shared data to avoid negative impacts from false-sharing.

DSM stacks often have to maintain data-copies at different nodes for performance reasons (e.g., readers on shared data) and therefore consistency semantics for shared data is

crucial for DSM performance. Maintaining consistent data-copies at all times (e.g., snooping processor caches) is costly/not-scalable due to network data movement and coordination overheads. Therefore, modern DSM stacks often support acquire/release[134, 135] consistency semantics capable of supporting sequential consistency for correctly synchronized applications.

Supporting PMEM aware DSM involves extending current acquire/release synchronization protocols with crash-consistency semantics. Hotpot [136] extends acquire/release semantics with redo-logging and supports single-writer , multiple-reader transactions with a two-phase commit like protocol. Hotpot evaluation is heavily biased towards applications designed for the block I/O APIs. Therefore, persistent DSMs for truly memory native applications remains as an open research question.

CHAPTER VIII

DISCUSSION, CONCLUSION AND FUTURE WORK

This dissertation addresses system software challenges associated with using new byte-addressable persistent memory (PMEM) hardware in server platforms. In the first half of this chapter we provide a discussion of the lessons learned in developing the solutions in our work. In the second half, we summarize our contributions and suggest possible important directions for future research.

8.1 Discussion

Our thesis work exploits the byte-addressability as the most important property of the new PMEM device, allowing it to be accessed as a traditional memory device, while at the same time providing persistence, like traditionally block-based storage devices. In our initial research work (§4 and §3), we advocated for replacing common file system-based I/O interfaces used in high-performance settings with memory-based APIs modeled after main memory object stores as a feasible I/O programming abstraction to the new persistent memory. Our evaluations prove that, extending memory-based I/O APIs with persistence semantics leads to greater opportunities for achieving improvements in performance and system efficiency than what can be afforded through new file system designs even when they are specialized for PMEM.

In the process, we also realized that it is possible to move away from I/O APIs altogether when designing applications for PMEM. This is because, PMEM being byte-addressable memory, eliminates the need for I/O software abstractions and intermediaries that were originally designed to bridge the impedance mismatch between applications' volatile (byte) and persistent (blocks) data access granularities. We benefit from our

PMEM programming experience in designing NVStream and PHX and introduce generalized, memory friendly programming primitives for PMEM native application programming in NVMTSX and Blizzard. Our NVMTSX work, introduces crash-sync-safe programming primitives for designing both single- and multi-threaded PMEM applications. Blizzard extends crash-sync-safe PMEM programming with fault-tolerance semantics. Combined together, NVMTSX and Blizzard, simplify application design, have positive impact on system performance, and more importantly, support additional application features and functionality with memory structures that are otherwise not feasible with block I/O programming.

Furthermore, our proposal and lessons learned in NVMTSX and Blizzard are useful in designing PMEM systems in a wide variety of application domains. Therefore, in hindsight, our PHX and NVStream work, in-turn could have benefited from being built upon the NVMTSX and Blizzard systems. For an example, the PMEM indexes of NVStream are likely to benefit from using persistent transactions developed with NVMTSX, and a fault-tolerant version of the same would benefit from Blizzard.

8.2 Lessons learned

We next present a summary of the general lessons we learned during this dissertation work.

8.2.1 The importance of system design optimized for persistent memory

Replacing existing block storage with fast PMEM devices (emulating block I/O) can significantly speedup traditional system storage software stacks. However, maximum performance gains require re-designing system implementation details such as caching, indexing or crash-consistency protocols with PMEM's 'memoryness' in mind. NVStream and PHX use PMEM optimized data-layouts, crash-consistent data updates and further improve system performance by exposing memory-friendly system APIs.

Next, from the PMEM-specific system building experience while developing NVStream

and PHX we gained important insights on how to provide generalized programming primitives for PMEM main memory programming. With the PMEM-optimized programming primitives provided through NVMTSX and Blizzard, persistent application state can be easily represented and manipulated using native PMEM structures. We learn that application designs that use proper PMEM primitives are more performant, can efficiently support system software properties such as high-availability, and, more importantly, enable feature-rich native memory data models (e.g. priority-queues). The later cannot be directly mapped to block-based storage abstractions, even when considering storage stack specialized for PMEM.

8.2.2 The importance of re-visiting the established system trade-offs

The mere introduction of memory-speed persistence of PMEM disrupts the well established assumptions in existing systems software. Porting only the storage software components to PMEM, in isolation, relative to the rest of the system, leaves much of the PMEM performance benefits untapped. Therefore, throughout this thesis work, we often had to step back and re-design our new system software solutions from the first principals of the system designs.

For an example, we had to revisit, how we do data replication in our effort to support highly available PMEM data-structure programming in Blizzard. Naively combining fast networking and fast PMEM libs did not produce necessary throughput due to networking overheads in the replication layer – an unlikely bottleneck point with traditional slow disk based replication. Removing the network bottleneck involved novel PMEM-aware zero-copy message broadcasts and message logging in the system.

Similarly, our PMEM specific optimizations for streaming HPC data storage in NVStream, opened up a seemingly new, PMEM specific bottleneck point – PMEM bandwidth. We solve the limited PMEM bandwidth challenge in our next work PHX by carefully using alternative system resources in HPC, namely the available interconnect bandwidth to peer

nodes. Our PHX proposal opens up opportunities to exercise new trade-offs among energy, interconnect bandwidths and DRAM capacity.

8.2.3 Benefits of optimizing across system abstractions

Separation of concerns is a best practice in software engineering. Grouping functionality together into coherent software components and forming new system software abstractions, encourages re-usability as the abstractions shield the developers from low-level complexities. However, putting abstract components together can incur overheads due to data-copying, serialization, redundant functionality and control path overheads.

In particular, optimizing across existing software abstractions is even more so important in the context of PMEM, as the device capabilities span across existing system software abstractions. For an example, the byte addressability of PMEM fits well under runtime and memory programming abstractions such as network buffers or memory APIs. The durability of PMEM works well with block I/O abstractions. We identify two main instances where we optimized system software for PMEM across standard abstraction boundaries during this thesis work.

- In Blizzard, we integrate the RAFT log, a block I/O abstraction, with network operations based on a memory abstraction, an eliminate the need for any extra data copies. Allocated PMEM buffers are shared between among the consensus and network system components using PMEM smart pointers.

- The NVStream and PHX designs introduce PMEM memory objects that are compatible with memory-based compute and checkpoint I/O (block-based) abstractions in HPC workflows. Our memory APIs remove unnecessary serialization overheads when moving data between compute and storage software components.

8.3 Summary

In the first part of this thesis, we analyze the system software overheads during HPC I/O data movements with PMEM as the backing storage medium. File I/O overheads of data serialization, kernel crossings and crash-consistent updates are significant during HPC checkpoints, even with PMEM optimized file-systems. We design and implement `NVStream`– a PMEM-aware I/O runtime that supports durable streaming I/O associated with HPC applications. `NVStream` uses a memory-based I/O interface that integrates with existing HPC I/O routines to accelerate persistent data writes. `NVStream` design leverage the streaming nature of I/O in HPC workflows, to benefit from using a log-structured PMEM storage engine design, that uses relaxed write orderings and append-only failure atomic semantics to form strongly consistent application checkpoints. `NVStream` I/O improves data movement time by up to 10$\times$ in real world scientific applications.

The second part of the thesis builds on the `NVStream` work to identify device bandwidth bottlenecks of PMEM hardware during HPC checkpoint I/O. The bulk, simultaneous data movements during application checkpoints stress out even the high-bandwidth memory interconnects of PMEM. We propose `PHX`– a PMEM backed, bandwidth aware I/O stack for application checkpoints. `PHX` uses alternative network data movement paths available in modern data centers to ease up the bandwidth pressure on the PMEM memory interconnects. We evaluate `PHX` against real world HPC application checkpoints. `PHX` improves GTC's checkpoint I/O times by up to 12$\times$ and cuts down the total simulation time by as much as 18%.

PMEM main memory can support both volatile data and persistent data, thus applications can use a single native memory data domain for both runtime and persistent state. However such a programming model includes maintaining ACID guarantees for the durable data of the application at all times. ACID-qualified persistent programming for multi-threaded applications is hard, as we have to reason about both crash-consistency and synchronization – crash-sync – semantics for programming correctness. In the third part of

the thesis we explore the design space of PMEM-aware durable transactional runtimes and their performance characteristics. We contribute new understanding of the correctness requirement for mixing different crash-consistent and synchronization protocols, characterize the performance of different crash-sync realizations for different applications and hardware architectures, and draw valuable insights for future system design choices.

In the fourth and final part of the thesis, we highlight the importance of reliable, highly-available PMEM programming. The characteristics of PMEM device change the currently established trade-offs in reliable system design which often are derived in the context of slow block I/O storage media. More importantly, PMEM changes the way we model our durable state of our applications. We design and implement `Blizzard`– a replicated persistent memory runtime that supports truly fault tolerant, concurrent and persistent data-structure programming. `Blizzard` carefully integrates userspace networking with byte addressable PMEM for a fast, persistent memory replication runtime. Additionally, `Blizzard`'s replication-aware crash-sync protocol enables concurrent data-structure updates all while maintaining consistent replica state.

8.4 Future work

The first generation commercial PMEM hardware has already hit the market [22], and an enhanced second generation was recently announced [137]. Going forward PMEM is expected to become mainstream in commodity servers. Efficient use of PMEM hardware will require further commitment from the systems research community. We present several promising future research directions directly influenced by the insights of this thesis work.

Redesigning applications for persistent memory : In the first part of this thesis we show how to accelerate HPC I/O with PMEM-aware I/O stacks. The technological advancements in PMEM technologies and device integration techniques (e.g., hardware caches) will further improve the access latencies of the PMEM device. Therefore, it is feasible to re-design

HPC application to use PMEM as main memory for both compute and I/O interactions. The scope of such work could exploring ideas such as:

- modeling simulation/analytic application data (volatile/durable) as PMEM main memory data-structures and exploring HPC-aware system software mechanisms for keeping them consistent under various failure models;
- exploring novel, memory friendly software abstractions (e.g., based on shared-memory) and optimizations for supporting cross-component interactions in HPC analytics pipelines.

Hardware support for next generation PMEM system software : Modern server systems integrate compute with I/O devices. One common example in high-end data centers is the presence of "smart", programmable network interfaces (NICs). Other examples are new, commercially available smart storage SSD devices integrating compute. These device-side compute elements often interact with main memory and offer accelerated services (e.g., hardware transactions, hardware packet stitching). With PMEM as main memory, we have to first, re-visit/re-design some of the existing hardware accelerated services designs and second, explore opportunities to introduce new hardware accelerated services that benefit from PMEM's byte addressable persistency.

Scalable, fault-tolerant PMEM aware application programming : Emerging classes of applications demand backend infrastructure services with fast access to massive storage capacity. These application backends are likely to grow in size with new user subscribes, products and services. Supporting these use cases with PMEM still has to be supplemented with scalable system software designs as the application data sizes far exceeds the node-local PMEM capacities. Promising research directions include:

- exploring distributed PMEM programming across data-shards (e.g., `Blizzard` shards) using memory native programming abstractions;

- exploring the feasibility of other distributed programming abstractions such as persistent distributed shared memory over PMEM.

Many other exciting systems research opportunities around persistent memory await at all levels of the system software stack and at the intersection of hardware-software. We hope such future efforts will benefit from the insights and the already open-sourced software-artifacts of this thesis work.

REFERENCES

[1] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, and S. R. Dulloor, "Basic Performance Measurements of the Intel Optane DC Persistent Memory Module," *arXiv preprint*, 2019.

[2] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory," *arXiv preprint arXiv:1908.03583*, 2019.

[3] Z. Tao, "Exadata with Persistent Memory: An Epic Journey," in *Programming Persistent Memory in Real Life (PIRL'19)*, Jul. 2019.

[4] *Available first on google cloud: Intel optane persistent memory*, `https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory`, 2020.

[5] P. Subrahmanyam, "Programming Persistent Memory in A Virtualized Environment using Golang," in *Programming Persistent Memory in Real Life (PIRL'19)*, Jul. 2019.

[6] *U.S. Department of Energy and Intel to deliver first exascale supercomputer*, `https://www.anl.gov/article/us-department-of-energy-and-intel-to-deliver-first-exascale-supercomputer`, 2019.

[7] S. Kannan, A. Gavrilovska, and K. Schwan, "Pvm: Persistent virtual memory for efficient capacity scaling and object storage," in *Proceedings of the Eleventh European Conference on Computer Systems*, ACM, 2016, p. 13.

[8] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*, ACM, 2014, p. 15.

[9] J. Xu and S. Swanson, "Nova: A log-structured file system for hybrid volatile/non-volatile main memories," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016, pp. 323–338.

[10] J. Arulraj, M. Perron, and A. Pavlo, "Write-behind logging," *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 337–348, 2016.

[11] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ACM SIGARCH Computer Architecture News*, ACM, vol. 39, 2011, pp. 91–104.

[12] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, Ontario, Canada, Oct. 2019.

[13] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," *ACM Sigplan Notices*, vol. 46, no. 3, pp. 105–118, 2011.

[14] H. Avni, E. Levy, and A. Mendelson, "Hardware transactions in nonvolatile memory," in *International Symposium on Distributed Computing*, Berlin, Heidelberg: Springer-Verlag, 2015, pp. 617–630, ISBN: 978-3-662-48652-8.

[15] T. Brown and H. Avni, "Phytm: Persistent hybrid transactional memory.," *PVLDB*, vol. 10, no. 4, pp. 409–420, 2016.

[16] Z. Wang, H. Yi, R. Liu, M. Dong, and H. Chen, "Persistent transactional memory," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 58–61, Jan. 2015.

[17] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Dhtm: Durable hardware transactional memory," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 452–465.

[18] K. Doshi, E. Giles, and P. Varman, "Atomic persistence for scm with a non-intrusive backend controller," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 77–89.

[19] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, and H.-L. Lung, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 465–479, 2008.

[20] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Redesigning lsms for nonvolatile memory with novelsm," in *2018 USENIX Annual Technical Conference*, 2018, pp. 993–1005.

[21] *Persistent memory development kit*, "https://pmem.io/pmdk/", 2020.

[22] *3d xpoint technology*, `https://www.micron.com/products/advanced-solutions/3d-xpoint-technology`, 2020.

[23] X. Wu and A. L. N. Reddy, "SCMFS: A File System for Storage Class Memory," in *SC*, 2011.

[24] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, and S. Klasky, "Goldrush: Resource efficient in situ scientific data analytics using fine-grained interference aware execution," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[25] C. Sewell, K. Heitmann, H. Finkel, G. Zagaris, S. T. Parete-Koon, P. K. Fasel, A. Pope, N. Frontiere, L.-t. Lo, and B. Messer, "Large-scale compute-intensive analysis via a combined in-situ and co-scheduling workflow approach," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–11.

[26] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible io and integration for scientific codes through the adaptable io system (adios)," in *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, 2008, pp. 15–24.

[27] R. Latham, *Parallel i/o in practice*, `http://www.nersc.gov/assets/Training/pio-in-practice-sc12.pdf`, 2012.

[28] C. Docan, M. Parashar, and S. Klasky, "Dataspaces: An interaction and coordination framework for coupled simulation workflows," *Cluster Computing*, vol. 15, no. 2, pp. 163–181, 2012.

[29] "Intel 64 and ia-32 architectures optimization reference manual," 2020.

[30] D. Otstott, N. Evans, L. Ionkov, M. Zhao, and M. Lang, "Enabling composite applications through an asynchronous shared memory interface," in *2014 IEEE International Conference on Big Data (Big Data)*, IEEE, 2014, pp. 219–224.

[31] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel, "Treadmarks: Distributed shared memory on standard workstations and operating systems," *Distributed Shared Memory: Concepts and Systems*, pp. 211–227, 1994.

[32] *Offset smart pointer for shared memory*, https://www.boost.org, 2018.

[33] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, "Optimizing checkpoints using nvm as virtual memory," in *IPDPS*, 2013.

[34] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ASPLOS '11*.

[35] *Gyrokinetic Toroidal Code*, `http://phoenix.ps.uci.edu/GTC/`.

[36] G Bryan and J Fritsch, "A benchmark simulation for moist nonhydrostatic numerical models," *Monthly Weather Review*, vol. 130, 2002.

[37] *How to emulate persistent memory*, `https://pmem.io/2016/02/22/pm-emulation.html`, 2018.

[38] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2017, pp. 135–148.

[39] A. Sasidharan and M. Snir, "Miniamr-a miniapp for adaptive mesh refinement," Tech. Rep., 2016.

[40] S. Lakshminarasimhan, N. Shah, S. Ethier, S. Klasky, R. Latham, R. Ross, and N. F. Samatova, "Compressing the incompressible with isabela: In-situ reduction of spatio-temporal data," in *European Conference on Parallel Processing*, Springer, 2011, pp. 366–379.

[41] J. C. Bennett, H. Abbasi, P.-T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, and V. Pascucci, "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, IEEE, 2012, pp. 1–9.

[42] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE Computer Society, 2010, pp. 1–11.

[43] *Hpc at exascale*, https://www.intel.com/content/www/us/en/high-performance-computing/supercomputing/exascale-computing.html, 2020.

[44] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "Plfs: A checkpoint filesystem for parallel applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, IEEE, 2009, pp. 1–12.

[45] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie, "Leveraging 3d pcram technologies to reduce checkpoint overhead for future exascale systems," in *SC '09*, ACM, 2009, p. 57.

[46] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson, "Moneta: A high-performance storage array architecture for next-generation, non-volatile memories," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE, 2010, pp. 385–395.

[47] S. Levy, B. Topp, K. B. Ferreira, D. Arnold, T. Hoefler, and P. Widener, "Using simulation to evaluate the performance of resilience strategies at scale," in *PMBS '13*, 2013, pp. 91–114.

[48] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, "Hybrid checkpointing using emerging nonvolatile memories for future exascale systems," *TACO*, vol. 8, no. 2, p. 6, 2011.

[49] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, "Optimizing checkpoints using nvm as virtual memory," in *IPDPS '13*, IEEE, 2013, pp. 29–40.

[50] D. Li, J. S. Vetter, G. Marin, C. McCurdy, C. Cira, Z. Liu, and W. Yu, "Identifying opportunities for byte-addressable non-volatile memory in extreme-scale scientific applications," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IEEE, 2012, pp. 945–956.

[51] *Summit supercomputer*, https://www.olcf.ornl.gov/summit, 2015.

[52] J. D'Ambrosia, "100 gigabit ethernet and beyond [commentary]," *IEEE Communications Magazine*, vol. 48, no. 3, S6–S13, 2010.

[53] *Stampede user guide*, https://portal.tacc.utexas.edu/user-guides/stampede.

[54] S. Gao, B. He, and J. Xu, "Real-time in-memory checkpointing for future hybrid memory systems," in *ICS '15*, ACM, 2015, pp. 263–272.

[55] S. Chakravorty and L. V. Kale, "A fault tolerant protocol for massively parallel systems," in *IPDPS '2004*, IEEE, 2004, p. 212.

[56] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, and J. Hiller, "Exascale computing study: Technology challenges in achieving exascale systems," *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, vol. 15, 2008.

[57] *Energy Consumption and Performance Design Space Trade-Offs for Optical Data Center Networks*, https://arpa-e.energy.gov/sites/default/files/Bergman_Columbia_Data_Center_Workshop_0.pdf.

[58] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak, "Intel® omni-path architecture: Enabling scalable, high performance fabrics," in *High-Performance Interconnects (HOTI), 2015 IEEE 23rd Annual Symposium on*, 2015.

[59] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers."

[60] J. H. Chen, A. Choudhary, B. De Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W.-K. Liao, K.-L. Ma, J. Mellor-Crummey, and N. Podhorszki, "Terascale direct numerical simulations of turbulent combustion using s3d," *Computational Science & Discovery*, vol. 2, no. 1, p. 015 001, 2009.

[61] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge, "Storage Management in the NVRAM Era," *PVLDB*, vol. 7, no. 2, pp. 121–132, 2013.

[62] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 133–146.

[63] A. M. Rudoff, *Deprecating the pcommit instruction*, https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction, 2016.

[64] *What's in HPE's persistent memory?* Retrieved from https://www.pcworld.com/article/3051133/whats-in-hpes-persistent-memory.html, 8 April 2016.

[65] *Big memory breakthrough for your biggest data challenges*, https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html, 2019.

[66] R. Kateja, A. Badam, S. Govindan, B. Sharma, and G. Ganger, "Viyojit: Decoupling battery and dram capacities for battery-backed dram," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ACM, 2017.

[67] M. Herlihy and J. E. B. Moss, *Transactional memory: Architectural support for lock-free data structures*, 2. ACM, 1993, vol. 21.

[68] N. Shavit and D. Touitou, "Software transactional memory," in *ACM Symposium on Principles of Distributed Computing (PODC)*, Aug. 1995, pp. 204–213.

[69] Intel Corporation, *Transactional Synchronization in Haswell*, Retrieved from http://http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/, 8 September 2012.

[70] *"intel architecture instruction set extensions programming reference"*, https://software.intel.com/, 2020.

[71] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood, "Supporting nested transactional memory in logtm," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006, pp. 359–370.

[72] S. Park, M. Prvulovic, and C. J. Hughes, "Pleasetm: Enabling transaction conflict management in requester-wins hardware transactional memory," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, IEEE, 2016, pp. 285–296.

[73] C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing," in *IEEE International Symposium on Workload Charaterization*, ser. ISPASS '08, Oct. 2008, pp. 35–46.

[74] E. Giles, K. Doshi, and P. Varman, "Brief announcement: Hardware transactional storage class memory," in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '17, New York, NY, USA: ACM, 2017, pp. 375–378, ISBN: 978-1-4503-4593-4.

[75] E. Giles, K. Doshi, and P. Varman, "Continuous checkpointing of htm transactions in nvm," in *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, ACM, 2017, pp. 70–81.

[76] J. Rao, E. J. Shekita, and S. Tata, "Using paxos to build a scalable, consistent, and highly available datastore," *arXiv preprint arXiv:1103.2408*, 2011.

[77] M. Poke and T. Hoefler, "Dare: High-performance state machine replication on rdma networks," in *Proceedings of the Symposium on High-Performance Parallel and Distributed Computing*, ACM, 2015, pp. 107–118.

[78] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the Annual Technical Conference*, USENIX Association, 2014, pp. 305–320.

[79] L. Lamport, "The part-time parliament," *ACM Transactions in Computer Systems*, vol. 16, no. 2, 1998.

[80] J. Gjengset, M. Schwarzkopf, J. Behrens, L. T. Araújo, M. Ek, E. Kohler, M. F. Kaashoek, and R. Morris, "Noria: Dynamic, partially-stateful data-flow for high-performance web applications," in *13th USENIX Symposium on Operating Systems Design and Implementation*, 2018, pp. 213–231.

[81] Z. István, D. Sidler, G. Alonso, and M. Vukolic, "Consensus in a box: Inexpensive coordination in hardware," in *Proceedings of the Conference on Networked Systems Design and Implementation*, USENIX Association, 2016, pp. 425–438.

[82] A. Kalia, M. Kaminsky, and D. Andersen, "Datacenter rpcs can be general and fast," in *Proceedings of the Symposium on Networked Systems Design and Implementation*, Boston, MA: USENIX Association, 2019, pp. 1–16.

[83] *Data plane development kit*, "https://www.dpdk.org", 2020.

[84] A. Klimovic, H. Litz, and C. Kozyrakis, "Reflex: Remote flash ≈ local flash," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2017.

[85] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler, "The scalable commutativity rule: Designing scalable software for multicore processors," in *Proceedings of the Symposium on Operating Systems Principles*, ACM, 2013, pp. 1–17.

[86] *Using persistent memory devices with the linux device mapper*, `https://pmem.io/2018/05/15/using_persistent_memory_devices_with_the_linux_device_mapper.html`, 2020.

[87] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[88] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, and P. Saab, "Scaling memcache at facebook," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation*, 2013, pp. 385–398.

[89] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, 2012, pp. 53–64.

[90] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation*, 2012, pp. 31–46.

[91] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ACM, 2013, pp. 472–488.

[92] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, "Chaos: Scale-out graph processing from secondary storage," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ACM, 2015, pp. 410–424.

[93] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "Llama: Efficient graph analytics using large multiversioned arrays," in *2015 IEEE 31st International Conference on Data Engineering*, IEEE, 2015, pp. 363–374.

[94] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" In *Proceedings of the 19th international conference on World wide web*, AcM, 2010, pp. 591–600.

[95] P. Kumar and H. H. Huang, "Graphone: A data store for real-time analytics on evolving graphs," in *17th USENIX Conference on File and Storage Technologies*, 2019, pp. 249–263.

[96] *Lobsters*, "https://lobste.rs/", 2019.

[97] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "Dudetm: Building durable transactions with decoupling for persistent memory," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 329–343, 2017.

[98] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, ACM, 2008, pp. 237–246.

[99] D. Dice, O. Shalev, and N. Shavit, "Transactional locking ii," in *International Symposium on Distributed Computing*, Springer, 2006, pp. 194–208.

[100] P. A. Bernstein and N. Goodman, "Multiversion concurrency controlâăt'theory and algorithms," *ACM Transactions on Database Systems (TODS)*, vol. 8, no. 4, pp. 465–483, 1983.

[101] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: Sql server's memory-optimized oltp engine," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 1243–1254.

[102] H. Kimura, "Foedus: Oltp engine for a thousand cores and nvram," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 691–706.

[103] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," in *ACM SIGPLAN Notices*, ACM, vol. 49, 2014, pp. 433–452.

[104] T. C.-H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster, "Nvthreads: Practical persistence for multi-threaded applications," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 468–482.

[105] D. Castro, P. Romano, and J. Barreto, "Hardware transactional memory meets memory persistency," *Journal of Parallel and Distributed Computing*, vol. 130, pp. 63–79, 2019.

[106] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14, Piscataway, NJ, USA: IEEE Press, 2014, pp. 265–276, ISBN: 978-1-4799-4394-4.

[107] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated persist ordering," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49, Piscataway, NJ, USA: IEEE Press, 2016, 58:1–58:13.

[108] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, Oct. 2014, pp. 216–223.

[109] M. Alshboul, J. Tuck, and Y. Solihin, "Lazy persistency: A high-performing and write-efficient software persistency technique," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2018, pp. 439–451.

[110] D. Narayanan and O. Hodson, "Whole-system persistence," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, England, UK, 2012, pp. 401–410, ISBN: 978-1-4503-0759-8.

[111] T. Wang and R. Johnson, "Scalable logging through emerging non-volatile memory," *Proc. VLDB Endow.*, vol. 7, no. 10, pp. 865–876, Jun. 2014.

[112] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46, New York, NY, USA: ACM, 2013, pp. 421–432, ISBN: 978-1-4503-2638-4.

[113] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via justdo logging," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, New York, NY, USA: ACM, 2016, pp. 427–442, ISBN: 978-1-4503-4091-5.

[114] V. J. Marathe, A. Mishra, A. Trivedi, Y. Huang, F. Zaghloul, S. Kashyap, M. Seltzer, T. Harris, S. Byan, B. Bridge, and D. Dice, "Persistent memory transactions," *CoRR*, vol. abs/1804.00701, 2018. arXiv: 1804.00701.

[115] B. M. Oki and B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, 1988, pp. 8–17.

[116] L. Lamport, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[117] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 335–350.

[118] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems.," in *USENIX annual technical conference*, vol. 8, 2010.

[119] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Symposium on Self-Stabilizing Systems*, Springer, 2011, pp. 386–400.

[120] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. Ports, "Building consistent transactions with inconsistent replication," *ACM Transactions on Computer Systems (TOCS)*, vol. 35, no. 4, pp. 1–37, 2018.

[121] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with cops," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 401–416.

[122] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[123] R. Love, *Linux kernel development*. Pearson Education, 2010.

[124] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The design and implementation of the 4.4 BSD operating system*. Addison-Wesley Reading, MA, 1996, vol. 2.

[125] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, and Y. Zhang, "H-store: A high-performance, distributed main memory transaction processing system," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1496–1499, 2008.

[126] K. Ren, A. Thomson, and D. J. Abadi, "Vll: A lock manager redesign for main memory database systems," *The VLDB Journal*, vol. 24, no. 5, pp. 681–705, 2015.

[127] K. Ren, A. Thomson, and D. J. Abadi, "Lightweight locking for main memory database systems," *Proceedings of the VLDB Endowment*, vol. 6, no. 2, pp. 145–156, 2012.

[128] J. M. Faleiro and D. J. Abadi, "Rethinking serializable multiversion concurrency control," *arXiv preprint arXiv:1412.2324*, 2014.

[129] J. M. Faleiro, A. Thomson, and D. J. Abadi, "Lazy evaluation of transactions in database systems," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 15–26.

[130] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, "Mojim: A reliable and highly-available non-volatile memory system," in *ACM SIGARCH Computer Architecture News*, ACM, vol. 43, 2015, pp. 3–18.

[131] T. Ma, M. Zhang, K. Chen, Z. Song, Y. Wu, and X. Qian, "Asymnvm: An efficient framework for implementing persistent data structures on asymmetric nvm architecture," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 757–773.

[132] U. Ramachandran and M. Y. A. Khalidi, "An implementation of distributed shared memory," *Software: Practice and Experience*, vol. 21, no. 5, pp. 443–464, 1991.

[133] M. Stumm and S. Zhou, "Fault tolerant distributed shared memory algorithms," in *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing 1990*, IEEE, 1990, pp. 719–724.

[134] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy release consistency for software distributed shared memory," *ACM SIGARCH Computer Architecture News*, vol. 20, no. 2, pp. 13–21, 1992.

[135] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Implementation and performance of munin," *ACM SIGOPS Operating Systems Review*, vol. 25, no. 5, pp. 152–164, 1991.

[136] Y. Shan, S.-Y. Tsai, and Y. Zhang, "Distributed shared persistent memory," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017, pp. 323–337.

[137] *Intel optane persistent memory 200 series announced*, `https://www.storagereview.com/news/intel-optane-persistent-memory-200-series-announced`, 2020.